Inferring Energy and Performance Cost of RTOS in Priority-Driven Scheduling

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 cost of Real-Time Operating Systems. We investigate priority-driven scheduling and assume inter-dependent tasks competing for shared resources.**

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 validate our approach by comparing it against gate level for accuracy and against TLM for speed. We obtain a worst-case error of 12% against a mean speedup of $\sim$30X.

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 often run in an instruction-set simulator, which is connected to the rest of the SoC TLM model. 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.

In our recent work [3] we have also shown how Funtime can factor in the impact of RTOS, with respect to both latency and energy. The approach is two-fold: first, we characterize the main components of a generic RTOS (context switches, clock tick interrupts, idle task, etc.) 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. In section IV we briefly summarize for clarity's sake how this is made possible. Second, 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. However, our previous case study was limited to independent tasks not competing for shared resources and scheduled using a Round Robin algorithm.

The contribution of the present paper consists in extending Funtime's RTOS estimation capabilities. In particular, we focus on the following aspects:

- *Target trace generation based on host profiling*: we enable the inference of the target execution trace by only properly profiling the application running on the host (PC). This is detailed in section V. In general, this step is necessary in order to correctly predict whether a task would get blocked due to inter-task dependency or resource contention. Detecting these conditions is critical as they strongly impact the variation in the overall execution time and, consequently, in the energy consumption as well.
- *Priority-driven scheduling prediction*: we extend the static analysis strategy that we proposed in [3] to predict the number of occurrences of the main OS components also in the case of priority-driven scheduling, and not

only Round Robin.

- *Resource contention*: we also account for the case of inter-dependent tasks competing for shared resources. These last two steps implicitly allow to estimate also the OS energy and timing impact. A detailed description is presented in section VI.

II. Related work

Yi et al. [4][5] 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 [6], 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 [5], [6] and the method proposed by us, Hessel et al. [7] 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 [8], 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 [9] 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 [5].

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 focus of this paper.

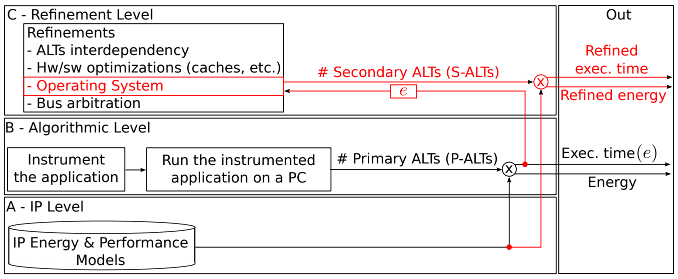


Fig. 1. The Funtime methodology flow (A-IP Level, B-Algorithmic Level, C-Refinement Level)

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 part of previous work and summarized for clarity's sake 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 the red box in 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].

Until now, Funtime was only able to infer the total number of times each target instruction was executed, but not their execution order. Enabling the inference of the actual execution order is however important to correctly determine the impact of inter-task dependence and resource contention on the RTOS energy and timing overhead. In section V we detail how this has been done.

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 the red box in 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. This is symbolically shown in Figure 1 and elaborated in the next sections. 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]; we have also partially factored in the OS-related overhead in [3] for the case of independent tasks not sharing any resource and scheduled using Round Robin. In the present paper, we further extend the OS refinement by accounting for inter-dependent tasks contending for shared resources and scheduled in a priority-driven fashion.

IV. RTOS characterization: summary

In this section, we summarize our previous work about how to characterize the S-ALTs due to the RTOS overhead in terms of latency and energy [3]. 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. When Funtime does the estimation instead, all the transactions are inferred, including the RTOS calls. The characterization process not only considers atomic RTOS calls, as it is done in [6], but also considers coarse-grained RTOS calls like clock tick interrupts, scheduler invocation, message queue broadcasting, sending and receiving. The number and type of atomic RTOS calls involved in these cases 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.

Although Table I shows a summary of characterized coarse-grained RTOS calls for the RTEMS RTOS [10] on a Leon-based SoC [11], the methods themselves are generic and do not depend on a specific RTOS implementation or SoC architecture. For each such RTOS call, 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. Only a subset of all the possible coarse-grained functionalities associated to an RTOS is listed here. This subset is sufficient to represent the case studies described in the next sections.

TABLE I
Energy and performance characterization for RTEMS on Leon3 (no cache)

RTOS calls	**# Leon3 S-ALTs**	**# Cycles**	σ **Cycles**	**Energy [nJ]**	σ **Energy**
clock tick interrupt	272	2241	24.00	80.14	0.96
scheduler + 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
msg q. broadcast to 3 + context switch	1510	12 263	6.94	434.21	7.77
msg q. send + context switch	834	6992	11.01	247.45	5.08
msg q. receive + context switch	830	6912	5.31	243.76	1.81

V. Target trace generation based on host profiling

This is the first contribution of the present paper. This step is critical for a correct implementation of our priority-driven scheduling prediction, which is detailed in section VI and is the second contribution of the paper.

All the steps reported in this section take place at the Algorithmic Level (Level B) of the Funtime methodology, as shown in Figure 1. We describe how a generic application can be profiled while executing on a common PC, i.e. the host, so to enable the inference of the actual instructions execution order for the target. Since at the IP Level (Level A) each target instruction has been previously characterized for energy and number of cycles, it becomes therefore possible to infer the actual advance in time after each instruction execution.

We redefine the previously-defined quantity e (see Figure 1) by splitting it in the following 2 sub-components:

1) e_{tot}: the total overall execution time of an application. This is the original value that Funtime was able to produce.
2) $e_{inc}(line)$: the incremental advance in time as a function of the source-code line number. This is the new quantity that Funtime can infer thanks to the enhancement proposed in this section and, in general, in this paper. The availability of $e_{inc}(line)$ is a key element of the RTOS refinement step in section VI, since it allows to

predict whether one or more RTOS tasks will be blocked because of inter-task dependency or resource contention.

The overall profiling process relies on the following 3 steps:

1) A generic application is compiled, profiled and executed on a common PC. Profiling of the executing application emits a string containing the file name, source code number and basic block number for each source line being executed. This allows to rebuild the exact source code execution trace. We recall that a basic block is a sequence of code with only one entry and one exit point.
2) Target mixed source/assembly files are generated for each source file of the same application. A mixed source/assembly file specifies the source line number corresponding to each basic block reported in the file. Or, the other way round, each basic block target instruction can be related to the proper source code line.
3) From 1 and 2, and given that within each basic block all the instructions are executed sequentially, the complete execution order for the target instructions can thus be inferred.

The application profiling described in step 1 can be enabled directly by the compiler at compilation time. Although in this section we detail how we have done this for the GNU GCC[1][12] compiler, the overall principle is general and compiler-independent.

A. *Enabling compiler-based profiling*

In a generic compiler, it is normally possible to identify at least 3 main components: a front-end, a middle-end and a back-end. The front-end is responsible for parsing the high-level language (HLL) source code files and generating a language-independent intermediate representation (IR) in form of control-flow graph. The output of the front-end is the input for the middle-end, which lowers the abstraction level down by applying a set of transformations and optimizations. The number of steps performed at this stage is highly dependent on the level of optimization requested and, in general, on the parameters passed at the compiler command line. Finally, the back-end has to translate the intermediate representation received from the middle-end into a machine-description (MD) file.

GNU GCC can also be described according to the above execution flow, as shown in Figure 2. The middle-end is itself composed of 3 different intermediate representations: GENERIC, which is a language-independent tree representation; GIMPLE, which is a reduced GENERIC subset used during optimization; RTL (Register Transfer Language), where the instructions to be output are defined in an algebraic form describing what the instruction does [13][14][15].

GNU GCC comes already by default with some profiling capabilities. In particular, the command-line switch `--coverage` adds a counter at the end of each basic block, so to enable a final total execution count of each source code line. Although this information provides a good code-coverage

[1]Version 4.4.1

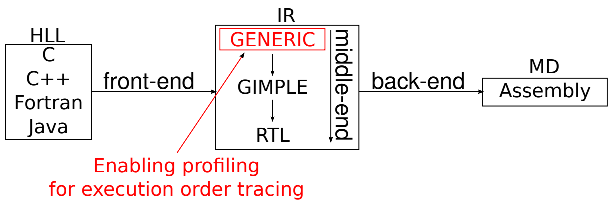
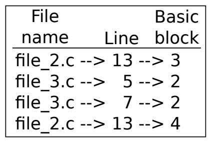


Fig. 2. The GNU GCC execution flow

overview, it only allows to infer the total number each target instruction is executed, but not the exact execution order. To overcome this limitation, we have extended the GCC profiling capabilities. In detail, at the GENERIC intermediate level, at the end of each basic block belonging to the application control-flow graph, we insert a call to a function that records the current file name, source line number and basic block number. This is symbolically shown in red in Figure 2. This allows to trace the exact source code execution order when the application is run on the host. In Figure 3 we show an example of the execution trace inferred.



File name		Line		Basic block
file_2.c	-->	13	-->	3
file_3.c	-->	5	-->	2
file_3.c	-->	7	-->	2
file_2.c	-->	13	-->	4

Fig. 3. Example of source execution trace

VI. Priority-driven scheduling prediction

In this section, which deals with the Refinement Level (Level C) of the Funtime methodology, we elaborate on how it is possible to build an algorithm to statically predict the occurrences of the RTOS functionalities in the case of priority-driven scheduling. To this purpose, a few new quantities are introduced here and categorized according to the Funtime abstraction level where their value is assigned.

User-defined parameters: we define the clock tick period as t_{tick}. We further define with $t_{rel,i}$ a task release time, i.e. the time when a task becomes available for execution. We then define with $\Pi(T_i)$ the priority level assigned to a generic task T_i. Finally we specify the source line numbers l_{sync}, within each task source code, where the task is supposed to have a synchronization point t_{sync}, such as accessing a message queue, a mutex and so on.

From Level A: we define a set of quantities corresponding to the execution time of some of the coarse-grain RTOS routines listed in Table I. For example, e_{sc} and e_{ct} are the execution time of a scheduler call and of a clock tick interrupt respectively. The quantity e_{del} is instead the execution time required to delete an ended task. An example is shown in Figure 4 – described in detail in *Case Study 1* – where the red bars correspond to e_{sc}, the blue bars to e_{ct} and the yellow bars to e_{del}. All these quantities, together with many more, are determined during the OS characterization phase, together with their respective energy consumption, as summarized in section IV.

From Level A and B: the quantity e first introduced in section III-B and then redefined in section V-A will from now on be referred to as e_i, and its sub-components $e_{tot,i}$ and $e_{inc,i}(line)$, being T_i a generic task. The actual number of tasks T_i is also provided here.

From Level C: we define a set of counters to count the occurrence of the RTOS routines involved in the analysis. For instance, SC, CT and DEL refer to the total number of calls to the scheduler, to clock tick interrupts and to the RTOS routines for tasks deletion when running an arbitrary number N of concurrent tasks.

Implementing an algorithm to predict SC, CT, DEL and, in general, the overall RTOS routines overhead, is the focus of this section and the main contribution of this work. Using the quantities introduced above, we detail how this can be done. To ease the explanation, we use 2 successively more realistic case studies. First, we assume the case of all independent tasks not competing for shared resources, but we allow different release times for the tasks, as well as the possibility that RTOS overheads may be a function of the number of ready tasks. Second, we keep the same assumptions as for the first example, except that we also allow inter-dependent tasks which can compete for shared resources. For a successful OS prediction in this second case study, we demonstrate the importance of the Funtime extension for host-profiling-based target trace generation described in section V. Note that, although in this paper we take RTEMS as the reference OS, the methodology is general and can be adapted to any OS.

A. Case study 1

In this case study, we consider the case where N independent tasks with priority $\Pi(T_1) > \Pi(T_i) > \Pi(T_N)$ have arbitrary release times $t_{rel,i}$ and where the execution time of some OS functionalities is dependent on the number of ready tasks $rdy(t)$ at the time t when such functionalities get triggered. For this reason, e_{sc}, e_{ct} and e_{del} are not constant figures, but depend on $rdy(t)$ and can be denoted as $e_{sc}(rdy(t))$, $e_{ct}(rdy(t))$ and $e_{del}(rdy(t))$ respectively. The availability of the $rdy(t)$ value also allows to infer the possible occurrence of the OS idle task, a condition that is true for $rdy(t) = 0$. For this case study, only the component $e_{tot,i}$ of the quantity e_i is needed.

Figure 4 is used as a reference during the algorithm description. We are here showing 3 independent tasks, being T_1 and T_3 the highest and lowest priority task respectively. At the figure bottom, we indicate the $t_{rel,i}$ and $t_{end,i}$ value associated to each task. Note that, while the value of $t_{rel,i}$ is user-assigned, the value of $t_{end,i}$ is calculated at run-time by the prediction algorithm. In addition, although $t_{rel,i}$ could in principle be assigned any time value, this value is seen and evaluated only at the occurrence of the next closest clock tick, when the scheduler is invoked. For this reason, we assume that either the Funtime user or a Funtime routine can take care of always setting a task $t_{rel,i}$ at a time which is a multiple of t_{tick}. The value of $t_{end,i}$ can instead occur at any time, since the scheduler is called by the OS routine that deletes the ended task. This is represented by the yellow bars in Figure 4.

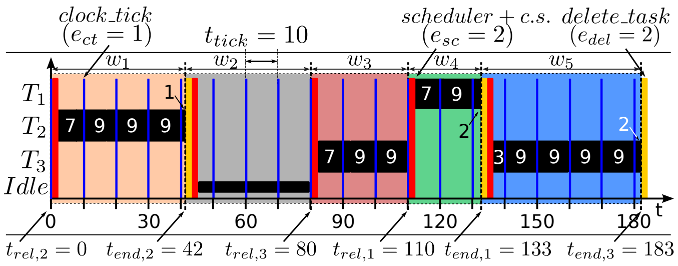


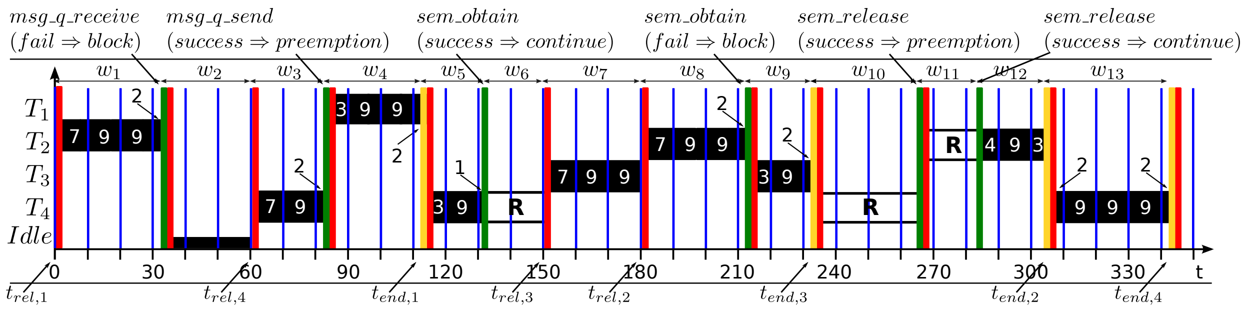
Fig. 4. 3 independent tasks T_1, T_2 and T_3. $t_{rel,i} \neq t_{rel,j}$

The value of $t_{rel,i}$ and $t_{end,i}$ is of great importance for the whole prediction algorithm, since evaluation is restricted to only these specific times. The reason is that there is no other time when we can encounter a variation in the tasks state or in the tasks ready queue. The result is a static decomposition of the whole scheduling into multiple time windows w_i. This is shown in Figure 4, 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, the following operations are performed.

1) the number of ready tasks $rdy(t)$ is calculated. If $rdy(t) = 0$, the OS Idle task is expected to run for the next time window w_i. This case is shown in Figure 4 at time $t = 42$.
2) the ready tasks are sorted according to their priority. The highest-priority task T_H is expected to run for the next time window w_i. For instance note that, as shown in Figure 4 at time $t = 100$, task T_3 is preempted by task T_1 that has higher priority.
3) the ending time of a time window is identified as the minimum time between the $t_{end,i}$ of the task running in that window and the next closest release time $t_{rel,i}$.
4) the values of SC, CT and DEL are updated with the related contributions given in the present window. We define such contributions as SC_w, CT_w and DEL_w.
5) This process is iterated until all tasks are completed.

The value of $t_{end,i}$, within each time window, is determined through the following considerations. First, the initial OS overhead for the window being estimated is taken into account and the current time $t_{current}$ is updated. For instance, when a new window starts after another task has ended, the OS routine for task deletion is always executed (yellow bar), followed by a call to the scheduler (red bar). Thus, if we take the beginning of window w_5 in Figure 4 as an example, we have that $e_{del} + e_{sc} = 2 + 2 = 4 \Rightarrow t_{current} = 133 + 4 = 137$. Second, we calculate the time difference between the next closest clock tick and the current time, we increment the current time of a quantity equal to the calculated time difference and we subtract that difference from the total task execution time $e_{tot,i}$. This is done to end up exactly at the time when a clock tick occurs and thus ease the calculation of the number



msg_q_receive (fail ⇒ block)
msg_q_send (success ⇒ preemption)
sem_obtain (success ⇒ continue)
sem_obtain (fail ⇒ block)
sem_release (success ⇒ preemption)
sem_release (success ⇒ continue)
w1 w2 w3 w4 w5 w6 w7 w8 w9 w10 w11 w12 w13
T1 T2 T3 T4 Idle
7 9 9
3 9 9
R
0 30 60 90 120 150 180 210 240 270 300 330 t
t_rel,1 t_rel,4 t_end,1 t_rel,3 t_rel,2 t_end,3 t_end,2 t_end,4

Fig. 5. 3 inter-dependent tasks T_1, T_2, T_3 and T_4. $t_{rel,i} \neq t_{rel,j}$

of total clock ticks happening until the end of the task. If we refer again to Figure 4, this step takes us from an initial $t_{current} = 137$ to a final $t_{current} = 140$. Third, the number of clock tick interrupts for the present window is calculated. We have previously defined this quantity as CT_w. It is true that $CT_w = \lceil \frac{e_{i,tot}}{t_{tick} - e_{ct}} \rceil$. Finally, $t_{end,i}$ can be calculated as $t_{end,i} = t_{current} + e_{tot,i} + e_{ct} \cdot CT_w$. For our example considering w_5, we have that $CT_w = \lceil \frac{38}{10-1} \rceil = 5$, and $t_{end,3} = 140 + 38 + 1 \cdot 5 = 183$, which is correct.

Note that, although our OS behavior prediction algorithm exhibits an iterative behavior, it still has an execution time proportional to the number of windows defined by the algorithm, and not to the tasks execution time $e_{tot,i}i$. Thus, it is still fast and clearly faster than running the actual OS.

B. Case study 2

In this realistic case study, we keep valid all the considerations made in *Case study 1*, but we also allow inter-dependent tasks that can compete for shared resources. Unlike the previous case study, here we also need the $e_{inc,i}(line)$ component of the quantity e_i.

For inter-dependent task, we mean a task that, at some point of its execution, reaches a synchronization point t_{sync} where it needs to wait for data from another task before being able to continue. As an example, consider Figure 5: at time $t = 33$, the highest priority task T_1 attempts to receive data from a message queue, but fails and gets blocked, since the queue is empty. At time $t = 83$, T_2 successfully sends some data into the same message queue and then gets preempted by T_1, that has now been unblocked and can thus resume its execution.

A shared resource could be a memory, a mutex or, in general, a serially-accessible resource. A conflict occurs when multiple tasks try to access the same resource at the same time. In general, the part of code dealing with accessing a resource is defined as *critical section* and normally it cannot be preempted, not even by a higher-priority task. This case is visible in Figure 5: at time $t = 132$, the lowest-priority task T_4 successfully locks a mutex and enters its critical section. The access to a shared resource R is shown as a white-filled rectangle. At time $t = 150$, T_4 is preempted by T_3 and, in turn, at time $t = 180$, T_3 is preempted by $T2$. Then at time $t = 213$, T_2 also tries to access resource R by locking the same mutex previously locked by T_4. However, the locking attempt fails and T_2 is blocked until T_4 releases the mutex at time $t = 265$.

This scenario is known as *priority inversion* and generally refers to the case of a high-priority task being blocked by a lower-priority task. Some techniques have been developed to avoid this problem. Two of them are known as *priority inheritance* and *priority ceiling*. We remind that the goal of this paper is not finding the best scheduling policy for a given scenario, but rather showing how, provided a priority-driven scheduling for a given scenario, Funtime can predict the OS components overhead. Nevertheless, compliance to the priority inheritance and priority ceiling techniques can be enabled in Funtime too, as part of the priority prediction algorithm. The user of Funtime will choose which policy to use.

In this *Case study 2* we account for the implications of having inter-dependent tasks that can compete for shared resources, since these factors can heavily affect the amount of OS overhead, i.e. number of context switches, occurrence and duration of the OS Idle task, etc. For example, it is only by knowing at which point in time T_1 tries to read from its message queue that we can estimate the duration of the OS Idle task. As another example, by knowing when T_4 tries to lock its mutex, we can predict whether T_4 will be able to get the lock before T_3 is released or not. If it does, we have the scenario shown in Figure 5 and described above. If not, T_4 will be able to lock its mutex only after T_2 and T_3 have finished. In this latter case, T_2 would not be blocked and therefore we would count 2 context switches less.

However, to be able to predict the time point t_{sync} when a task tries to access an OS resource, like a message queue or a mutex (see the top of Figure 5), the only knowledge of the total task execution time $e_{tot,i}$ is not enough any more. Instead, the following 2 conditions must also be satisfied. Note that enabling these satisfiability conditions is one of the contributions of this paper:

1) it must be possible to evaluate the advance of time with the granularity of a single source-code line. This quantity has been defined as $e_{inc,i}(line)$ and can be retrieved by instrumenting the compiler as described in section V.
2) the Funtime user must specify at what source-code line l_{sync} the synchronization point occurs and what dependency relation exists.

From the 2 points above, it results that $t_{sync} = e_{inc}(l_{sync})$.

As in *Case study 1*, the proposed OS prediction algorithm relies on taking decisions at specific time points, which correspond to the end of temporal windows w_i. The only difference is that not only $t_{rel,i}$ and $t_{end,i}$ are used to identify a window edges, but also the time points where a task reaches a synchronization point. This is shown in Figure 5, where we identify 13 windows.

In case of inter-dependent tasks, in order to run an application natively at Level B (Algorithmic Level), it is necessary that the Funtime user manually provides a value to the variables that, at Level C (Refinement Level), would be assigned at a synchronization point. We recall that Level B is the only level where an application is run in Funtime (see Figure 1). In case that an application takes different branches depending on the value received, the Funtime user chooses the values according to the specific needs, so to have a trace of *Primary* ALTs available for the different possibilities. Later on, he/she can also specify a percentage value corresponding to the frequency each branch is taken. Note instead that, at Level C, no application is run at all, as well as no Operating System.

Algorithm 1 reports the pseudo-code for the main steps described in *Case study 1* above. The text in red color distinguishes instead the extra component needed to account for *Case study 2*.

Algorithm 1

```
/* Initialize all the OS quantities to zero */
$SC(rdy(t)) = CT(rdy(t)) = \cdots = DEL(rdy(t)) \leftarrow 0$;
total execution time left $e\_tot\_left \leftarrow \sum_{i=1}^{N} e_i$;
current time $t_{current} \leftarrow 0$;
while $e\_tot\_left > 0$ do
    for $i$ = 1 to N do
        $rdy(t) \leftarrow$ number of ready tasks at time $t$;
        $T_H(t) \leftarrow$ highest priority task ready at time $t$;
    end for
    calculate $T_H$ ending time $t_{end,T_H}$;
    $t_{next} \leftarrow$ min($t_{end,T_H}$, closest $t_{rel}$, closest $t_{sync}$);
    /* Update all the OS quantities */
    update $SC(t_{next} - t_{current}, rdy(t))$;
    update $CT(t_{next} - t_{current}, rdy(t))$;
    ⋮
    update $DEL(t_{next} - t_{current}, rdy(t))$;
    update $e\_tot\_left$;
    $t_{current} \leftarrow t_{next}$
end while
```

VII. OS REFINEMENT VALIDATION

A. Accuracy: Funtime vs. gate level

Accuracy validation is performed by comparing energy and timing results inferred by Funtime to those extracted from back-annotated gate-level simulation, which is taken as a reference. The reason is that gate level is very accurate.

The SoC platform used for validation is the same used for characterization and is shown in Figure 6. Communication is implemented by the AMBA AHB/APB bus, while computation relies on the SPARC-based Leon3 processor, configured without cache. The entire system runs at the frequency of 40 MHz.

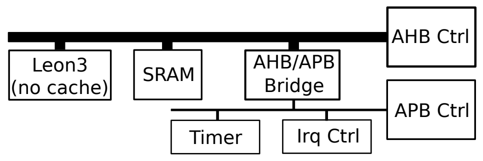


Fig. 6. Leon3-based platform

The goal is to verify the Funtime prediction accuracy for an RTEMS-based signal-processing software composed of 5 inter-dependent tasks, as shown in Figure 7. Task T_1 provides a data stream of 100 elements. Such a stream is broadcast to T_2, T_3 and T_4 through the message queue Q_1. Always, only one of the 3 tasks T_2, T_3 and $T4$ performs some processing on the received data, depending on the actual data value. The processed data is then sent into the message queue Q_2 and reaches the output process T_5. This modeling flow could be for instance representative of an equalizer.

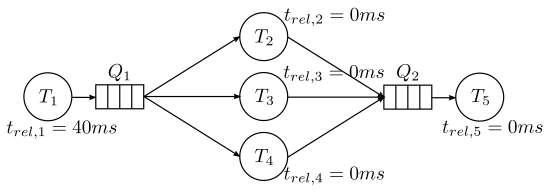


Fig. 7. RTEMS-based signal-processing software

The RTEMS clock tick period t_{tick} has been set to 200μs. The task priority has been assigned as follows: $\Pi(T_5) > \Pi(T_2) > \Pi(T_3) > \Pi(T_4) > \Pi(T_1)$. This means that T_5 and T_1 have the highest and lowest priority respectively. This is done since we expect that the system is immediately ready to operate whenever the input data stream arrives. In this case, the stream arrives when T_1 is released, i.e. at $t_{rel,1} = 40ms$, which is an integer multiple of t_{tick}. The release time for the remaining tasks is instead $t_{rel,2...5} = 0ms$.

We have collected the results in Table II, which is itself split into 3 horizontal subtables considering successively more sources of inaccuracies as we move down. Subtable 1 only considers the error deriving from the OS characterization summarized in section IV, while $[e_1, ..., e_5]$, as well as the OS activities are measured from gate level. Subtable 2 has two sources of inaccuracies: the OS characterization and the OS activity prediction, described in section IV and VI respectively, while $[e_1, ..., e_5]$ 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 $[e_1, ..., e_5]$. Table II is further split vertically into a left and a right side: the left side reports figures related to the five tasks $[T_1, ..., T_5]$, independently of the OS; the right side is instead meant to show the OS overhead. The energy values shown refer to the Leon3 processor. Both real and estimated values are shown along with the percentage error.

TABLE II
VALIDATING FUNTIME VS. GATE LEVEL FOR ENERGY ESTIMATION: PRIORITY-DRIVEN SCHEDULING

	Applications						**RTEMS OS**							
	$\mathbf{T_1}$	$\mathbf{T_2}$	$\mathbf{T_3}$	$\mathbf{T_4}$	$\mathbf{T_5}$		**clock ticks**		**msg q. broadcast(3) + c.s.**		**msg q. send + c.s.**		**msg q. receive + c.s.**	
Subtable 1 (1 source of inaccuracy: OS characterization)														
Real e_i [ms]	0.3	11.3	5.1	14.0	9.8	Real #OS macro-func.		1174		100		100		400
Real en. [μJ]	0.5	15.8	7.1	19.5	13.7	Real en. [μJ], time [ms]	89.0	61.6	44.1	31.4	25.8	18.3	99.0	71.5
						Inferred en. [μJ], time [ms]	94.1	65.7	43.4	30.7	24.7	17.5	97.5	69.1
						Error [%]	+5.7	+6.6	-1.5	-2.4	-4.0	-4.6	-1.5	-3.4
Subtable 2 (2 sources of inaccuracy: OS characterization + OS activity prediction)														
Real e_i [ms]	0.3	11.3	5.1	14.0	9.8	Real #OS macro-func.		1174		100		100		400
						Inferred #OS macro-func.		1263		100		100		400
						Error [#]		+89		0		0		0
Real en. [μJ]	0.5	15.8	7.1	19.5	13.7	Real en. [μJ], time [ms]	89.0	61.6	44.1	31.4	25.8	18.3	99.0	71.5
						Inferred en. [μJ], time [ms]	101.2	70.7	43.4	30.7	24.7	17.5	97.5	69.1
						Error [%]	+13.7	+14.8	-1.5	-2.4	-4.0	-4.6	-1.5	-3.4
Subtable 3 (3 sources of inaccuracy: OS characterization + OS activity prediction + $\mathbf{e_i}$ value)														
Real e_i [ms]	0.3	11.3	5.1	14.0	9.8	Real #OS macro-func.		1174		100		100		400
Inferred e_i [ms]	0.3	9.8	5.0	12.8	8.1	Inferred #OS macro-func.		1234		100		100		400
Error [%]	-5.4	-13.0	-1.2	-8.5	-16.3	Error [#]		+60		0		0		0
Real en. [μJ]	0.5	15.8	7.1	19.5	13.7	Real en. [μJ], time [ms]	89.0	61.7	44.1	31.4	25.8	18.3	99.0	71.5
Inferred en. [μJ]	0.5	14.8	7.6	19.3	12.2	Inferred en. [μJ], time [ms]	98.9	69.1	43.4	30.7	24.7	17.5	97.5	69.1
Error [%]	-6.3	-5.9	+6.2	-1.4	-10.5	Error [%]	+11.1	+12.0	-1.5	-2.4	-4.0	-4.6	-1.5	-3.4

The RTEMS functionalities that we account for in Table II are, from left to right, the clock tick interrupt, the message queue broadcast (to 3 queues), the message queue send and the message queue receive. The inferred number of clock ticks is higher than the real one. The reason is that the real value of t_{tick} was measured to be in average $\sim$218μs instead of the ideal 200μs. The inferred number of the remaining RTOS routines is instead equal to the real one. The reason is that their occurrence is directly visible from the inferred execution trace, and is independent of the real or inferred task execution time e_i. In general we get good accuracy figures both for energy and timing estimation, within the 12% of the real value.

B. Speedup: 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 speedup achieved by Funtime over TLM-PV simulation is $\sim$30X. This confirms the capacity of Funtime to be used for complex and real use-case scenarios.

VIII. CONCLUSIONS AND FUTURE WORK

We have presented a high-level method for rapidly and accurately estimating energy and performance overhead of Real-Time OS. While in our previous work we only addressed Round Robin scheduling and assumed independent tasks, in this work we have enhanced our investigation by considering priority-driven scheduling and assuming inter-dependent tasks competing for shared resources. 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 ($\sim$30X) compared to TLM-PV, 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 the case of multi-processing systems.

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] S. Penolazzi, I. Sander, and A. Hemani, “Predicting Energy and Performance Overhead of Real-Time Operating Systems,” in *DATE*, Dresden, Germany, 2010.
[4] Y. Yi, D. Kim, and S. Ha, “Virtual Synchronization Techniques with OS Modeling for Fast and Time-accurate Cosimulation,” 2003.
[5] ——, “Fast and accurate cosimulation of mpsoc using trace-driven virtual synchronization,” *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 2007.
[6] C. Brandolese and W. Fornaciari, “Measurement, Analysis and Modeling of RTOS System Calls Timing,” in *Euromicro*, 2008.
[7] 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.
[8] M. A. Hassan, K. Sakanushi, Y. Takeuchi, and M. Imai, “Enabling RTOS Simulation Modeling in A System Level Design Language,” in *ASP-DAC*, 2005.
[9] Z. He, A. Mok, and C. Peng, “Timed RTOS Modeling for Embedded System Design,” in *RTAS*, 2005.
[10] “RTEMS Homepage. http://www.rtems.com.”
[11] “AEROFLEX GAISLER. http://www.gaisler.com.”
[12] “GNU Compiler Collection. http://gcc.gnu.org.”
[13] J. Merrill, “Generic and gimple: A new tree representation for entire functions,” in *Proceedings of the GCC Developers Summit*, 2003.
[14] *GNU Compiler Collection Internals*.
[15] A. Vichare, *The Conceptual Structure of GCC*, Indian Institute of Technology, Bombay, 2008.