Predictive OS Modeling for Host-Compiled Simulation of Periodic Real-Time Task Sets

Parisa Razaghi and Andreas Gerstlauer, *Senior Member, IEEE*

***Abstract*—With the increasing complexity of embedded software, host-compiled simulators have been introduced to address the need for a fast simulation environment. However, designers pay the price for higher performance with a loss in timing accuracy. In this letter, we introduce a novel predictive OS model to provide fast software simulation with accurate scheduling of periodic real-time tasks. The OS model predicts the next preemption point by monitoring system state, and automatically and optimally adjusting the granularity of back-annotated delays. We evaluated our simulator on a range of periodic task sets. Our observations show that we can achieve the same 99% accuracy as a simulation at 1 μs granularity with an average 230x speedup.**

***Index Terms*—Host-compiled simulation, RTOS modeling.**

I. Introduction

With an ever increasing fraction of complex embedded software, efficient evaluation of software execution on a target architecture at early stages of the design process is a crucial part of today's system-level design methodologies. Transaction level modeling (TLM) approaches based on system level design languages (SLDL), such as SystemC, provide high-level hardware/software (HW/SW) cosimulation backplanes. However, integrating traditional instruction set simulators (ISS) into TLM backplanes is no longer appropriate for simulation of software execution due to their slow simulation speeds.

Recently, host-compiled or source-level approaches have received widespread attention as solutions that can provide both fast and accurate simulations [1]–[4]. In such approaches, the actual application code is natively compiled and executed on the simulation host to achieve the fastest possible functional simulation. To model timing, the code is back-annotated with execution delay estimates. Finally, back-annotated code is wrapped into an abstract model of the OS [5], [6] and software execution environment [7], [8] integrated into a proprietary or standard-based SLDL and TLM cosimulation backplane. Such processor models have been shown to simulate at speeds beyond 500 MIPS with more than 95% timing accuracy.

In HW/SW cosimulation approaches, higher speed is achieved by coarse-grained simulation of the system, which inherently comes at a loss in timing accuracy. Several researchers have focused on improving the accuracy of high-level simulators while maintaining similar performance. Krause *et al.* [9] present combined ISS and abstract RTOS model cosimulation. This approach replaces an actual RTOS binary code with an abstract model running outside the ISS and performs cycle-accurate thread switches. Khaligh *et al.* [10] present an adaptive TLM simulation kernel, which changes the level of accuracy during simulation to the level expected by designers. Schirner *et al.* [11] introduce a granularity-independent approach for accurate simulation of interrupts on host-compiled processor models by applying optimistic prediction and correction. In all cases, however, fundamental speed and accuracy tradeoffs remain. At higher levels of abstraction, real-time task simulators allow early evaluation of real-time performance of a given task set [12], [13], but are based on idealized task models that do not allow for execution of actual task functionalities in a complete system context.

Manuscript received October 07, 2011; accepted January 23, 2012. Date of current version March 23, 2012.

P. Razaghi and A. Gerstlauer are with the Department of Electrical and Computer Engineering, The University of Texas at Austin, Austin, TX 78712 USA (E-mail: parisa.r@mail.utexas.edu, gerstl@ece.utexas.edu).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/LES.2012.2186281

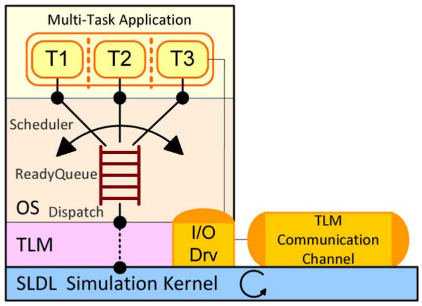

Fig. 1. Host-compiled software simulator.

By contrast, this letter presents a novel RTOS modeling approach for accurate and fast evaluation of real-time periodic task sets in a host-compiled HW/SW cosimulation context. Traditionally, the assumption is that timing errors are bounded by annotated discrete timing granularities. However, this turns out not to be the case when modeling preemptive behavior. The contributions of this letter are twofold: (1) an analysis of error bounds in preemption models that shows that errors can exceed limits set by timing granularities is presented; and (2) a simple extension to existing RTOS models that is able to avoid such errors and provide an accurate simulation of scheduling and preemption effects is proposed. The proposed RTOS model predicts the next preemption point by monitoring the system state and optimally adjusting the granularity of back-annotated delays in the application code.

II. Host-Compiled Software Simulation

Fig. 1 outlines the host-compiled software simulator, details of which can be found in [14]. We follow a layered approach to construct the simulator. Running on top of an SLDL simulation kernel, a TLM layer provides a high-level interface to the rest of the system and connects to a standard TLM backplane. An OS model manages the scheduling, queuing, dispatch, and execution of application tasks according to an emulated scheduling policy, ensuring that at any given time only one task is active

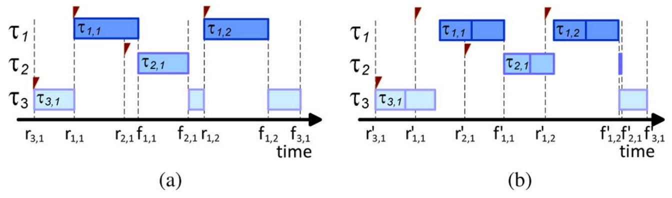
τ1
τ2
τ3
τ1,1
τ2,1
τ3,1
τ1,2
r3,1 r1,1 r2,1 f1,1 f2,1 r1,2 f1,2 f3,1
r'3,1 r'1,1 r'2,1 f'1,1 r'1,2 f'1,2 f'2,1 f'3,1
time
(a)
(b)

Fig. 2. Error models in host-compiled simulation. (a) Theoretical order. (b) Simulation order.

on the underlying TLM/SLDL kernel. Finally, the user application is modeled as a set of sequential and concurrent high-level C/C++ processes. It sits on top of and accesses services of the OS layer via a canonical OS API.

For the following discussion, we focus on analyzing preemptive scheduling behavior. Without loss of generality, we consider an ideal model that abstracts away other effects, where an application is composed out of a set of periodic real-time tasks $\Gamma = \{\tau_1, \tau_2 \cdots \tau_n\}$, ordered by decreasing priorities P_i. Each task τ_i is described by its period T_i and execution time C_i. The j^{th} instance (job) of task τ_i is denoted by $\tau_{i,j}$, and its response time $R_{i,j}$ is measured as the time elapsed between its release time $r_{i,j}$ and the time $f_{i,j}$ when it finishes execution of one iteration. For host-compiled simulation, task execution delays are assumed to be modeled by timing values back-annotated at a granularity of $\delta_i (C_i = \delta_i * n)$. To model preemptions, the OS scheduler is called at the end of each δ_i interval (or when a task makes an explicit OS kernel call).

We can analyze the accuracy of host-compiled simulation by evaluating the response time of each task and measuring the percentage of error using the following equation:

$$\mathrm{err}_i = \frac{|R_i(\mathrm{model}) - R_i(\mathrm{ideal})|}{R_i(\mathrm{ideal})}$$

Fig. 2(a) shows the theoretical execution of three periodic tasks in which the first job $\tau_{3,1}$ of task τ_3 is preempted by the higher-priority job $\tau_{1,1}$ at times $r_{1,1}$ and $r_{1,2}$. While τ_1 is in its first iteration $\tau_{1,1}$, a medium priority task τ_2 is released (at time $r_{2,1}$) and gets executed once $\tau_{1,1}$ finishes (at time $f_{1,1}$). Subsequently, $\tau_{3,1}$ resumes its execution only after both higher priority jobs finish (at time $f_{2,1}$).

In Fig. 2(b), the host-compiled simulation of the same task set is shown. In this model, execution delays of tasks are divided into discrete intervals. Since the OS scheduler is only called at the end of each advance in time, the start of $\tau_{1,1}$ is delayed until the end of the current time interval of $\tau_{3,1}$. Consequently, the start of $\tau_{2,1}$ is also shifted by an equal amount to the delayed time $f'_{1,1}$ at which $\tau_{1,1}$ finishes. As a result, the next job of τ_1 now starts in the last time interval of $\tau_{2,1}$, and $\tau_{2,1}$ gets preempted by $\tau_{1,2}$ as soon as its last time interval expires. Therefore, $\tau_{2,1}$ cannot finish executing its last block of code, complete its job, and return control to the OS kernel until it is resumed at time $f'_{1,2} = f'_{2,1}$ when $\tau_{1,2}$ finishes.[1] In other words, $f'_{2,1}$ is now determined by $f'_{1,2}$.

As shown in this example, we can consider three sources of errors in response times. In a first scenario, the start time of a job of a higher priority task τ_i is delayed by a lower priority task τ_j running at its release time. The largest delay is achieved when the higher priority job is released simultaneously with the start of a new time interval of the lower priority job.

[1]Note that if the last code block is instead moved to before the last time advance, a job can conversely finish too early by an equal amount.

As such, the maximum error can be bounded from above by $\mathrm{err}_i \leq \max_{j=i+1}^{n}(\delta_j)/C_i$.

In a second scenario, the start time of a job of a lower priority task $\tau_{j,l}$, which is released when a job of a higher priority task $\tau_{i,k}$ is running, is determined by the finish time of the higher priority job. Hence, if the start time of $\tau_{i,k}$ is shifted due to scenario one or two, the start time of $\tau_{j,l}$ is shifted equally. As such, the maximum error in this scenario is equal to the maximum error derived for scenario one or two of all higher priority tasks: $\mathrm{err}_j \leq \max_{i=1}^{j-1}(\mathrm{err}_i) = \max_{i=1}^{n}(\delta_i)/C_j$.

Finally, the third scenario happens when the start time of $\tau_{j,l}$ is delayed due to scenario one or two such that a job of a higher priority task $\tau_{i,k}$ becomes active while $\tau_{j,l}$ is executing its last time interval, as is the case for $\tau_{2,1}$ in Fig. 2(b). In this scenario, the finish time of $\tau_{j,l}$ is determined by the finish time of all higher priority tasks released while any higher priority task is running, i.e., until $\tau_{j,l}$ can get resumed. As such, the amount of error depends on the system load and, assuming a well-behaved system, is only limited by the task's next deadline $\mathrm{err}_j \leq (T_j - C_j)/C_j$.

As demonstrated by this analysis, the error in simulated task response times in scenarios one and two is a function of the modeled timing granularity. However, the possibility of large errors in the third scenario severely limits host-compiled HW/SW cosimulators for evaluating real-time performance.

III. Predictive RTOS Model

In the following, we propose a novel predictive abstract RTOS model for fast yet accurate host-compiled simulation of periodic real-time task sets. In this model, all sources of preemption errors are eliminated and the designer does not need to be concerned with selecting a proper granularity for task delays. The key idea is to optimally and automatically adjust the timing granularity by predicting the next scheduling point based on the states of application tasks at any given time.

A. RTOS Model Architecture

Fig. 3 shows the structure of the abstract RTOS model, which replicates a typical OS architecture. The RTOS model consists of four internal queues that maintain the state of tasks running on the processor. A *Ready* queue holds tasks that are ready to execute and is sorted based on a user-defined scheduling policy. An *Idle* queue holds periodic tasks that have called the kernel's `TaskEndCycle()` method at the end of their iteration. The *Idle* queue is ordered based on the release time of each task's next iteration. *Idle* tasks are retrieved from the head of the queue and placed in the *Ready* queue by the OS kernel at the start time of their next period. Tasks waiting for an event are suspended and transferred to a *Wait* queue upon calling a `PreWait()` method. The blocked task will be placed back in the *Ready* queue when a `PostWait()` method is called to release it. Finally, a *Sleep* queue holds tasks that have been suspended until they are resumed again.

At the core of the OS model is the OS scheduler, which is invoked by the OS API methods whenever a context switch is possible or required. It blocks the currently running task and dispatches the task at the top of the *Ready* queue to execute. In addition, the OS kernel advances simulation time whenever the currently running task calls a `TimeWait()` method. In conventional OS models, the granularity of delays is defined by the application code and the scheduler is called after advancing the simulation time to allow for preemption of the current task by any available higher priority task. As described in Section II,

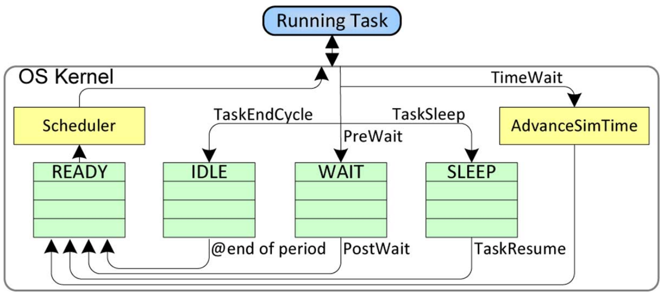

Fig. 3. Abstract RTOS model.

```
Function PredictNextPreemptionTime (task runningTask):
for tasks in Idle Queue do
    if idleTask::Priority ≥ runningTask::Priority then
        predictedTime := idleTask::nextPeriod - currentTime
        return PredictedTime
    endif
endfor
```

(a)

```
Function TimeWait (long long nsec, task runningTask):
 1  remainedDelay := nsec
 2  while remainedDelay > ø do
 3      adjustedDelay := PredictNextPreemptionTime(runningTask)
 4      if ( !Empty(Wait) ) then
 5          adjustedDelay := defaultDelayGranularity
 6      endif
 7      if adjustedDelay > remainedDelay then
 8          adjustedDelay := remainedDelay
 9      endif
10      remainedDelay := remainedDelay - adjustedDelay
11      AdvanceSimTime(adjustedDelay)
12      Scheduler()
13  endwhile
```

(b)

Fig. 4. Timing granularity adjustment. (a) Preemption point prediction. (b) Time delay.

this limits the possibility of accurately modeling preemptions. By contrast, our predictive OS model automatically adjusts the granularity of user-defined delays in order to call the scheduler at actual preemption points.

B. Predictive Scheduler

The key idea behind removing the preemption error is to predict the next possible preemption point and invoke the scheduler at the proper time. Fig. 4(a) shows the algorithm for predicting the next preemption time in a system running a set of periodic tasks. Since the *Idle* queue is sorted based on the tasks' next release times, the preemption point is defined by the first task with a priority higher than the currently running task.

Fig. 4(b) shows the pseudo code of the `TimeWait()` method in the predictive OS model. It first computes the adjusted time delay by calling the method to predict the next preemption point (line 3). Since the exact preemption point is unknown whenever a task is waiting for an external event, the OS kernel, in this case, falls back to a user-defined default timing granularity (lines 4–6). After advancing the simulation time by the adjusted delay (line 11), the OS scheduler is called to perform a context switch and block the current task until it is scheduled again (line 12). This loop continues until the user-defined delay is consumed. As can be seen, the designer does not need to settle on a granularity for back-annotated delays. The OS kernel itself breaks delays into a number of smaller steps as needed, in order to automatically provide the best timing granularity for fully accurate results.

IV. Experimental Result

We simulated a set of randomly generated artificial periodic tasks and compared the simulation performance of our OS model to a conventional model under different timing granularities. Accuracy is analyzed by comparing results to the execution of tasks on a reference ISS [15] modeling a singlecore MIPS Malta platform running a Linux 2.6 kernel configured with preemption and high resolution timers.

The experimental setup consists of randomly generated periodic tasks with uniformly distributed periods over $[1, 100]$ ms and task weights $w_i = C_i/T_i$ over $[0.001, 0.1]$ for small (S), $[0.1, 0.4]$ for large (L), and $[0.001, 0.4]$ for medium (M) tasks. The priority of tasks are assigned inversely to their periods following a ratemonotonic scheduling scheme. The execution delay of tasks is modeled by a delay loop of no-operation (NOP) instructions. We ran each task set for 10 s of simulated time. At a nominal rate of 100 MIPS simulated by the reference ISS, this corresponds to 1000 million NOP instructions. Task sets have been generated to cover various task weight ranges under different total system utilizations $U = \sum w_i$.

We analyzed the accuracy of our approach by comparing the response times of periodic tasks in the reference ISS with our host-compiled simulator. Delays were back-annotated into host-compiled models directly from measurements taken when running on the ISS. Model error was measured as the average absolute difference in individual task response times over all tasks and task iterations. Table I summarizes the task set properties and compares the accuracy and performance of our predictive RTOS (P-RTOS) model with that of a conventional model at four different back-annotation granularities. We can observe that the highest possible accuracy is achieved using the P-RTOS model. This is equivalent to a conventional model at 1 μs granularity, which loses a large amount of accuracy at coarser granularities. Note that although we would expect to see zero errors on the predictive model, our previous experience has shown [14] that remaining errors are caused by OS contextswitch overheads and nonideal behavior of a real Linux system not included in our RTOS model. In terms of simulation performance, an average simulation speed of 67 GIPS is achieved on the P-RTOS model. This is 233 times faster than the original OS model at a granularity of 1 μs and similar to the original model at 1 ms granularity.

In the conventional OS model, designers are responsible for choosing the timing granularity to achieve acceptable accuracy and performance. However, selecting the proper granularity is not straightforward. For example, using the granularities of 1 μs and 10 μs, the same accuracy is provided while the former simulates 10 times faster than the latter. In addition, the lack of a reference platform for many applications makes it impossible to find a reliable granularity. Fig. 5 plots the tradeoff between average accuracy and simulation speed over all task sets. As can be seen, decreasing the timing granularity results in higher accuracy but comes at a loss in simulation performance. By contrast, our predictive model provides both fast and accurate results regardless of the timing granularity.

In order to evaluate our approach under realistic conditions with HW/SW interactions, we also simulated a task set composed out of a subset of applications from the automotive category of the MiBench suite [16]. Benchmarks were converted to execute periodically and concurrently based on rate monotonic scheduling policy, where task Susan(edge) was modified to interact with an FPGA by streaming its outputs over the system bus. The resulting task set was simulated for 500 s both on the

TABLE I
ARTIFICIAL PERIODIC TASK SET CHARACTERISTICS AND SIMULATION RESULTS

Task Set	**S1**	**S2**	**S3**	**S4**	**S5**	**M1**	**M2**	**M3**	**M4**	**L1**	**L2**	**L3**
Number of Tasks	7	11	9	12	13	4	4	4	3	3	3	3
Avg. Task Weight	0.047	0.043	0.062	0.058	0.064	0.136	0.173	0.176	0.286	0.21	0.212	0.294
CPU Utilization	0.33	0.47	0.56	0.70	0.84	0.54	0.70	0.71	0.86	0.63	0.64	0.89
Avg. Err. (1μs)	0.53%	0.48%	0.48%	0.79%	0.88%	0.41%	0.08%	0.45%	0.08%	0.17%	0.18%	0.14%
Avg. Err. (10μs)	0.54%	0.53%	0.49%	0.82%	0.89%	0.42%	0.08%	0.44%	0.08%	0.18%	0.18%	0.13%
Avg. Err. (100μs)	0.95%	1.63%	0.96%	3.47%	2.98%	0.83%	0.11%	0.95%	0.13%	0.21%	0.28%	0.34%
Avg. Err. (1000μs)	5.77%	7.34%	5.57%	15.9%	12.8%	5.02%	0.32%	0.99%	1.22%	1.77%	0.90%	2.41%
Avg. Err. P-RTOS	0.53%	0.48%	0.48%	0.79%	0.88%	0.41%	0.08%	0.45%	0.08%	0.17%	0.18%	0.14%
Speed [GIPS] (1μs)	0.55	0.36	0.31	0.25	0.20	0.33	0.25	0.25	0.21	0.28	0.27	0.19
Speed [GIPS] (10μs)	4.5	3.6	2.8	2.3	2.1	2.8	2.3	2.4	1.9	2.4	2.4	2.0
Speed [GIPS] (100μs)	20	20	25	17	6	14	14	14	12	12	17	10
Speed [GIPS] (1000μs)	100	50	50	50	25	100	100	100	500	100	50	50
Speed [GIPS] P-RTOS	107	32	33	19	18	62	107	54	85	99	103	87

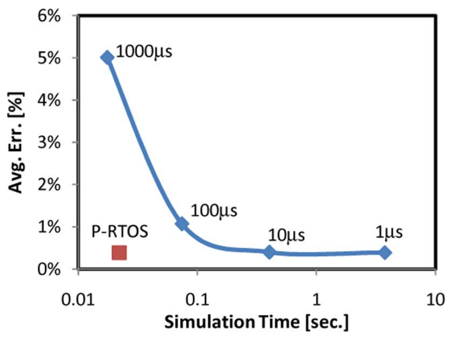

Fig. 5. Accuracy and speed trade-offs.

TABLE II
SIMULATION RESULTS FOR AUTOMOTIVE TASK SET

Task	**Exec. Delay**	**Period**	**Weight**
susan(edge)	1.36s	4.7s	0.29
susan(smooth)	3.50s	38s	0.09
qsort	1.15s	45s	0.03
basicmath	37.26s	85s	0.44
Simulation	**Avg. Err.**	**Max. Err.**	**Speed**
RTOS-1μs	0.20%	26.7%	115 MIPS
RTOS-1ms	1.44%	26.8%	820 MIPS
RTOS-10ms	1.64%	27.0%	825 MIPS
RTOS-100ms	3.45%	28.5%	830 MIPS
P-RTOS	0.006%	0.022%	840 MIPS

reference ISS and in host-compiled form (with back-annotated ISS delays). Table II summarizes benchmark features and compares the accuracy and the simulation performance for the predictive and the conventional RTOS models. Assuming a nominal CPI of 1 and not counting idle cycles, the P-RTOS model simulates at 840 MIPS. This is 7 times faster than the conventional model at 1 μs, where a granularity of 100 ms is required to achieve the same speed. However, the third error scenario is triggered in this setup, leading to high average and maximum errors in conventional models at any granularity. By comparison, P-RTOS model accuracy remains above 99%.

V. SUMMARY AND CONCLUSIONS

In this letter, we presented a predictive RTOS model designed for host-compiled software simulation of real-time periodic task sets. The simulator automatically adjusts the granularity of back-annotated delays by predicting the next task preemption point. Our model combines very fast simulation speed with error-free scheduling, which makes host-compiled simulators suitable for rapid, early evaluation of the real-time performance of periodic task systems within a HW/SW cosimulation context. In this work, we focused on solutions for avoiding errors in the preemption model. We have started to integrate this approach into our full host-compiled processor model and system simulator, which support sporadic tasks and inter- and intra-processor task communications [17]. In the future, we plan to include models of cache, pipeline, and other dynamic effects that influence preemption costs, task execution times and hence, accuracy of overall real-time scheduling behavior.

REFERENCES

[1] J. Schnerr, O. Bringmann, A. Viehl, and W. Rosenstiel, "High-performance timing simulation of embedded software," *DAC*, Jun. 2008.
[2] J. Ceng, W. Sheng, J. Castrillon, A. Stulova, R. Leupers, G. Ascheid, and H. Meyer, "A high-level virtual platform for early MPSoC software development," CODES + ISSS, Sep. 2009.
[3] K. Lin, C. Lo, and R. Tsay, "Source-level timing annotation for fast and accurate TLM computation model generation," *ASP-DAC*, Jan. 2010.
[4] T. Meyerowitz, A. Sangiovanni-Vincentelli, M. Sauermann, and D. Langen, "Source-level timing annotation and simulation for a heterogeneous multiprocessor," *DATE*, Mar. 2008.
[5] H. Posadas, J. A. Adamez, E. Villar, F. Blasco, and F. Escuder, "RTOS modeling in SystemC for real-time embedded SW simulation: A POSIX model," *DAES*, vol. 10, no. 4, Dec. 2005.
[6] J. C. Prevotet, A. Benkhelifa, B. Granado, E. Huck, B. Miramond, F. Verdier, D. Chillet, and S. Pillement, "A framework for the exploration of RTOS dedicated to the management of hardware reconfigurable resources," *Reconfigurable Computing and FPGAs*, 2008.
[7] A. Bouchhima, I. Bacivarov, W. Yousseff, M. Bonaciu, and A. Jerraya, "Using abstract CPU subsystem simulation model for high level HW/SW architecture exploration," *ASPDAC*, Jan. 2005.
[8] G. Schirner, A. Gerstlauer, and R. Dömer, "Fast and accurate processor models for efficient MPSoC design," *TODAES*, vol. 15, no. 2, Feb. 2010.
[9] M. Krause, D. Englert, O. Bringmann, and W. Rosenstiel, "Combination of instruction set simulation and abstract RTOS model execution for fast and accurate target software evaluation," *CODES+ISSS*, Oct. 2008.
[10] R. S. Khaligh and M. Radetzki, "Modeling constructs and kernel for parallel simulationof accuracy adaptive TLMs," *DATE*, Mar. 2010.
[11] G. Schirner and R. Dömer, "Introducing preemptive scheduling in abstract RTOS models using result oriented modeling," *DATE*, Mar. 2008.
[12] F. Singhoff, J. Legrand, L. Nana, and L. Marce, "Cheddar: A flexible real time scheduling framework," *ACM SIGAda Ada Letters*, vol. 24, no. 4, pp. 1–8.
[13] RTSIM: Real-Time Simulator [Online]. Available: http://rtsim.sssup.it
[14] P. Razaghi and A. Gerstlauer, "Host-compiled multicore RTOS simulator for embedded real-time software development," *DATE*, Mar. 2011.
[15] OVP: Open Virtual Platform [Online]. Available: http://www.ovpworld.org
[16] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A free, commercially representative embedded benchmark suite," presented at the WWC Dec. 2001.
[17] P. Razaghi and A. Gerstlauer, "Automatic timing granularity adjustment for host-compiled software simulation," *ASPDAC*, Jan. 2012.