Accurate Timed RTOS Model for Transaction Level Modeling

Yonghyun Hwang
CECS
University of California
Irvine, CA, 92617
Email: yonghyuh@uci.edu

Gunar Schirner [1]
Dept. of ECE
Northeastern University
Boston, MA, 02115
Email: schirner@ece.neu.edu

Samar Abdi [1]
Dept. of ECE
Concordia University
Montreal, Canada H3G 1M8
Email: samar@ece.concordia.ca

Daniel G. Gajski
CECS
University of California
Irvine, 92617
Email: gajski@uci.edu

***Abstract*—In this paper, we present an accurate timed RTOS model within transaction level models (TLMs). Our RTOS model, implemented on top of system level design language (SLDL), incorporates two key features: RTOS behavior model and RTOS overhead model. The RTOS behavior model provides dynamic scheduling, inter-process communication (IPC), and external communication for timing annotated user applications. While the RTOS behavior model is running, all RTOS events, such as context switch and interrupt handling, are passed to RTOS overhead model to adopt the overhead during system execution. Our RTOS overhead model has processor- and RTOS-specific pre-characterized overhead information to provide cycle approximate estimation. We demonstrate the applicability of our model using a multi-core platform executing a JPEG encoder. Experimental results show that the proposed RTOS model provides the high accuracy, 7% off compared to on-board measurements while simulating at speeds close to the reference C code.**

I. Introduction

Recent design paradigm is rapidly shifting to platform based design with intensive use of software. In such a platform based design, choosing an optimal platform and a best mapping of application to platform is crucial to meet performance and real-time constraints of an embedded system. Aforementioned system level decisions can be guided by the accurate analysis of system performance for a given design choice. For practical performance analysis, fast and accurate estimation of software performance with timing accurate RTOS modeling is one key solution for rapid design space exploration and early prototyping. To accurately estimate performance of software, three delay contributors have to be modeled: (a) D_{exec}, for execution of application code; (b) D_{sched}, due to dynamic scheduling (e.g. delay due to task preemption); and (c) D_{rtos}, the RTOS overhead to provide RTOS services, such as task management, task scheduling, task synchronization, and interrupt handling.

Significant research efforts have been made to estimate D_{exec}. [8] and [1] can take into account the datapath structure which adopt a bus functional model and integrate ISS. They provide accurate results at the expense of speed. To obtain more speed while providing accurate estimation, [3] and [11] provide fast system simulation models. However, they are not retargetable which, in turn, limits scalability. Abstract RTOS models have been developed on top of System Level Design Languages (SLDLs) (e.g. SystemC [6], SpecC [4]) to expose the effect of dynamic scheduling, D_{sched}. Examples include [5], modeling typical RTOS primitives on top of SpecC, and [10], implementing an POSIX API on top of SystemC. The later offers an interesting combination of online estimation and RTOS-modeling, which enables an optimized modeling of periodic interrupts. However, the solutions do not include accurate modeling of RTOS overheads that are addressed in the paper.

In this paper, accurate timed RTOS model for transaction level modeling (TLM) is presented which focuses on the estimation 'accuracy' along with the estimation 'fidelity'. For the estimation accuracy, the presented RTOS model reflects two major delays, D_{sched} and D_{rtos}. D_{sched} is exposed by supporting all the RTOS primitive operations, such as task scheduling, interrupt handling and so on. D_{rtos} is revealed by RTOS overhead model which has processor- and RTOS-specific pre-characterized overhead information. For D_{exec}, application estimation technique proposed in [7] is used. Estimation technique [7] analyzes the application codes and annotates the timing information back to the codes. These annotated application codes are integrated into our timed RTOS model to simulate D_{exec}.

This paper is organized as follows. Sec. II outlines the timed RTOS model consisting of RTOS behavior model and RTOS overhead model. In Sec. III, experimental results show accuracy, speed, and scalability using industrial scale design like JPEG application. Finally, we conclude the paper and touch on future works.

II. Timed RTOS Model

The timed RTOS model is divided into two separate models: RTOS behavior model and RTOS overhead model. RTOS behavior model captures fundamental RTOS services, such as task management, event handling for synchronization, OS management, time modeling, and interrupt handling. In addition to fundamental RTOS services, it supports abstract channels and core OS system calls to provide communication among tasks, which allows easy integration with legacy application codes. The RTOS overhead model has processor- and RTOS-specific pre-characterized overhead information

[1]The work presented in this paper was done while authors were affiliated with CECS at UCI

978-3-9810801-6-2/DATE10 © 2010 EDAA

to provide cycle approximate estimation. While the RTOS behavior model is running, all RTOS events are passed to RTOS overhead model to analyze the timing overhead in system execution. In following sub-sections, RTOS behavior and overhead model will be presented.

A. RTOS Behavior Model

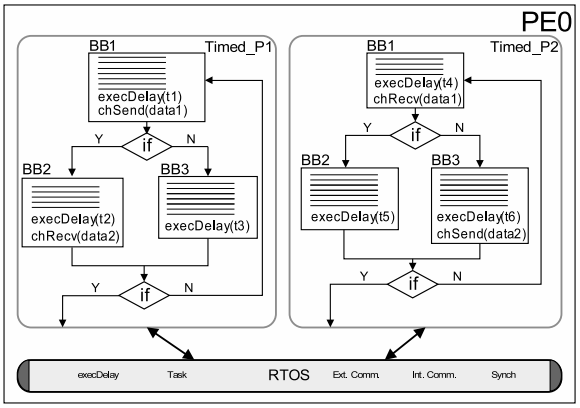

Fig. 1. RTOS Behavior Model

Fig. 1 shows our RTOS behavior model that emulates fundamental RTOS services. The RTOS behavior model is implemented as a SC_INTERFACE inside the SC_MODULE of the PE. Each of virtual processes (i.e., a time annotated process) is implemented as a SC_THREAD for multiple flows of execution. The execution of each SC_THREAD is controlled by the model through SystemC event to emulate the selected scheduling policy.

To emulate scheduling, the RTOS behavior model is very similar to an actual RTOS. It maintains a Task Control Block (TCB) for each virtual processes and the appropriate queue for task scheduling. Each virtual process is scheduled based on a task state machine where a transition is made by the scheduling policy. At any given point in time only one SC_THREAD (one virtual process) is released through the SystemC event based on scheduling policy.

At the beginning of timed RTOS model execution, the proposed model parses RTOS configuration script describing scheduling policy and processes' priorities. After parsing step is complete, it initializes internal data structures for scheduling and puts all the virtual processes in READY state. Then, it calls scheduler to pick up the first runnable virtual process, changing its READY state to a RUNNING state, and executes it. While the virtual process is running, it calls an wrapper API of the RTOS model at the end of every basic blocks to simulate progress of time due to code execution. Our virtual processes, having time annotated code, calls this interface (*execDelay()*, Fig. 1), and subsequently a *sc_core::wait()* is executed under RTOS control. During this time, the virtual process remains in the RUNNING state and no other process is scheduled.

During the time progress, an incoming interrupt may release a higher priority process in WAITING state which is pending on a resource. This calls scheduler. The scheduler makes the state of the higher priority process as a RUNNING state and makes the current running process READY. Please note that the above description is for priority based scheduling. If scheduling policy is round robin, progress of time is checked against allocated time slot that sits in RTOS configuration script. If the allocated time slot for the virtual process is expired, scheduler makes the state of the current process as READY and starts execution of the available process from the queue.

The RTOS behavior model furthermore provides an abstract channels to enable internal communication and external communication to the outside of the processor. To realize external communication in the model, an abstract bus model [12] is adopted, which is implemented on top of interrupt handling service of our model to synchronize external communication among hardware. As for internal communication, SystemC event is used instead of interrupt handling service to achieve synchronized communication among virtual processes.

Abstract channels, supported by our RTOS model, can trigger scheduling. To give an example, when a virtual process executes an inter-process transaction (e.g. Fig. 1, call *chRecv()* in process *Timed_P2*), this transaction is executed under RTOS control. Assuming it contains acquiring a non-available semaphore, the virtual process state is set to PENDING, the next process is selected from the ready queue according to the scheduling policy, set to RUNNING, and released through its SystemC event. Finally, the old virtual process suspends by waiting its own SystemC event. Note that the actual context switch is performed by SystemC kernel and the RTOS behavior model only controls their release.

B. RTOS Overhead Model

Modeling RTOS overhead is challenging because it depends on type of RTOS, CPU, and platform. It gets more complicated if RTOS, CPU, and platform are configurable which is common in a modern embedded system. One option for the modeling is to do static source code analysis. However, the analysis is complicated by a limited source code availability especially for a commercial RTOS, as well as API and structural/organizational differences between implementations. These factors inhibit a static source code analysis or make it prohibitively expensive. Another option is to use data hand book provided by RTOS vendor. Even though it is convenient to use, it provides the fidelity instead of the accuracy in RTOS overhead estimation.

In order to model RTOS overhead accurately without RTOS source code, we have developed a time stamping approach on RTOS API level. From observed time stamps, we derive a set of overheads for RTOS primitive operations such as scheduling overhead and task synchronization overhead. Because the RTOS overhead is sensitive to RTOS and CPU, we characterize a RTOS on the actual processor in supported configuration(s). The following paragraphs describe our analysis approach.

Several special test applications are developed to capture time stamps. They invoke RTOS primitives in a controlled environment in which we know a priority of the scheduling

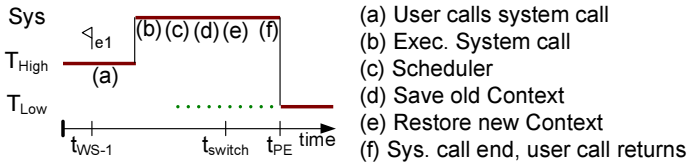

Fig. 2. sem_take() with context switch

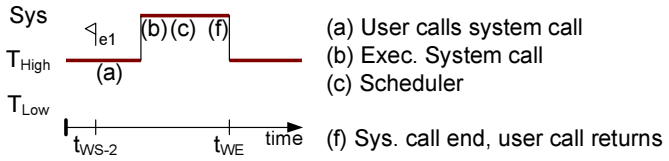

Fig. 3. sem_take() without context switch

outcome. The time stamp codes and its data are exclusively placed in a non-cached fast local memory to minimize impact on execution time. Timer interrupts are also disabled while analyzing timing unrelated RTOS primitives to eliminate the impact of unexpected interrupts.

In this paper, we show one example on how the scheduling overhead can be obtained. In Fig. 2 and Fig. 3, the example of our test application is shown based on acquiring a semaphore without and with a context switch. As system calls basically follow the same steps, the example is representative.

Fig. 2 shows the case with context switch. Before system call, sem_take, the time stamp, t_{WS-1} is recorded in user mode. After time stamping, task T_{High} calls *sem_take()* to acquire non-available semaphore (a), which results in a context switch to T_{Low}. In side of system call, *sem_take()*, the processor mode is switched to kernel mode, and the actual semaphore code is executed (b) after looking up a system call table. As semaphore is not available, the scheduler (c) determines the next task to execute, the current task's context is saved (d), the new task's context is restored (e). Finally, system call return (f) in the new task's context. For system call return, the processor mode is switched back to user mode and RTOS executes T_{Low}

In the application, T_{Low} had earlier relinquished the CPU by posting a semaphore to T_{High}. Therefore, the system call return point in T_{Low} is in a *sem_post()*. After exiting *sem_post()*, we put another time stamp t_{PE}. Please note, that the code for returning a system call is independent of the call type (e.g. the code is identical for *sem_post()* and *sem_take()* starting at the point for restoring new context (c)). We can therefore use t_{PE} to analyze the duration of a *sem_take()* with the scheduling overhead.

Fig. 3 illustrates the case of an available semaphore where no context switch necessary. We put the time stamp, t_{WS-2}, before the system call, *sem_take()*. The sequence during the system call is short cut. Because the task, T_{High}, can get semaphore, scheduling is not necessary (c). Therefore, there is no context switch and the system call returns (f) to the same task where we record t_{WE}. At this point, we can calculate the duration of a *sem_take()* without the scheduling overhead.

Based on these analysis, we can determine the duration for *sem_take()* and the scheduling. Our analysis with these time stamps, after eliminating the overhead of time stamping itself, is as follows:

$$\begin{aligned} Dur(sem\_take) &= t_{WE} - t_{WS-2} \\ Dur(sched) &= t_{PE} - t_{WS-1} - Dur(sem\_take) \end{aligned}$$

The duration analysis of *sem_take* is the difference between start and end time stamp. We compute the scheduling duration based on the duration of the *sem_take* with a context switch ($t_{PE} - t_{WS-1}$) and subtract the time for *sem_take()* without context switch. We chose *sem_take()*, since we expect portion (b), execution of the system call itself, to be minimally dependent on the scheduling outcome as *sem_take()* only manipulates the own task state. Conversely, *sem_post()* shows a variance beyond the estimated context switch duration as it manipulates other task's states.

We analyze the delays for other RTOS primitives, such as communication and synchronization primitives, in a similar way. For each primitives, we measure the time without and with the scheduling overhead. After normalizing for the already determined scheduling duration, we calculate the average between the two cases to determine the primitive's duration.

In addition to the fundamental RTOS primitives, we also characterized core OS system calls as explained earlier. In particular, we have characterized *memcpy()* as it is frequently used and its code is often heavily optimized. We measure the delay for *memcpy()* by putting two time stamps before and after *memcpy()* with varying data size. We keep the delay results in a table, and use a linear extension to estimate values beyond the table boundaries.

We store the analyzed RTOS characteristics in our database, with a separate delay for each used RTOS primitives. The code for instantiating an RTOS is created automatically during TTLM generation. The selected RTOS' characteristics are retrieved from the database based on the system specification. With the characteristics, small SystemC script is automatically generated to initialize the selected RTOS model. During the execution of an our RTOS model, the characterized delay – without scheduling – (e.g. Dur(*sem_take()*)) is executed for the specific RTOS primitive. To give an example, our abstract task dispatcher, switching between SC_THREADs, is annotated with $Dur(switch)$. We use this basically state less delay model to simplify abstract RTOS implementation while maintaining a high simulation performance. Even though our analysis and modeling approach abstracts away many influences on RTOS overhead (e.g. number of total, waiting tasks, manipulated tasks), it already yields valuable feedback for estimating system performance.

III. Experimental Results

To evaluate the benefits of our approach, we modified ESE tool [2] and have applied it to industry size design, a JPEG encoder. Our JPEG encoder has 6 tasks. Two tasks are used for input and output for JPEG encoding algorithm. JPEG encoding algorithm likely consists of 4 tasks, DCT, quantization, zigzag and huffman encoding. JPEG encoding

Granularity	Board Measure	RTOS		Timed RTOS	
		Est.	Error	Est.	Error
64	33.3M	18.7M	-43.84%	30.4M	-8.87%
32	34.9M	19.0M	-45.43%	31.4M	-9.93%
16	35.9M	19.4M	-45.89%	33,3M	-7.34%
8	38.3M	20.0M	-47.55%	36.8M	-3.83%
Average	**N/A**	**N/A**	**-46%**	**N/A**	**-7%**

TABLE I
ESTIMATION ACCURACY OF MODELS

	Model Sim. Time			
Granularity	**64**	**32**	**16**	**8**
RTOS	0.68 s	0.70 s	0.70 s	0.73 s
Timed RTOS	0.69 s	0.74 s	0.76 s	0.79 s

TABLE II
SIMULATION TIME OF MODELS

algorithm, all the 4 tasks, is mapped to the same processor, a Microblaze with 100Mhz, and scheduled by Xilinx's RTOS, Xilkernel. Input provider and output consumer, are assigned to custom hardware components. To realize communication between Microblaze and custom hardware, we introduced Transducer through which all the data transfers are relayed. The following paragraphs discuss our results on accuracy, speed, and scalability of our estimation engine using the above experimental setup.

Table I shows the average accuracy of our abstract models for each of our designs in comparison to cycle-accurate execution on the Xilinx FF896 board. To illustrate the impact of adding overhead information to RTOS model, we analyze two abstraction levels, TTLM with RTOS and TTLM with timed RTOS. TTLM with RTOS adds a behavior RTOS model to TTLM (Sec. II). In contrast to TTLM with RTOS, TTLM with timed RTOS additionally models system overheads (Sec. II-B), reflecting D_{exec}, D_{sched}, and D_{rtos}. Our results show that modeling D_{exec} and D_{sched} is not sufficient for parallel applications. The TTLM with RTOS shows 46% average error, up to 48% depending on data transfer granularity. With the fine grained IPC, the designs exhibit a significant system overhead and thus the TTLM with RTOS underestimates by 46% on average. Adding RTOS overhead modeling reduces the error to less than 10%, yielding already sufficiently accurate timing information. The remaining error is due to our abstract analysis and modeling of RTOS overheads, which we chose in favoring automate ability and simulation speed. Comparing solutions, our TTLM with Timed RTOS yields the most accurate timing estimation. With increasing system overhead, an even smaller error is shown. For the smallest granularity, size 8, our generated TLM exhibited only 3.83% error.

Table II summarizes the simulation time of our model in number of seconds. The results show that TTLM with RTOS model simulates in fractions of a second. Our TTLM with timed RTOS also simulates in fractions of a second. No significant increase in speed is measurable to reflect RTOS overheads. These results clearly demonstrate the advantages of our solution, simulating fast enough to do early design exploration, while exhibiting the accuracy close to the board measurement.

Generation time is an additional usability aspect on scalability. Generation time for TTLM with timed RTOS is dominated by the application timing annotation engine [7], as it executes the Low Level Virtual Machine (LLVM [9]) compiler. The additional effort for instantiating the timed RTOS is negligible because our RTOS model is implemented as a library and small scripts are the only needs for RTOS configuration. The measured total generation time is around 1.2 seconds for JPEG encoder algorithm, which shows our timed RTOS model is scalable.

IV. CONCLUSIONS

This paper presents accurate timed RTOS model for TLMs. The proposed RTOS model fulfills three important requirements, accuracy, speed, and scalability by providing three key aspects: (a) dynamic scheduling through RTOS behavior emulation, (b) modeling of RTOS overheads, and (c) interrupt modeling for external communications. Our model enables fast performance evaluation with high accuracy and exposes performance bottlenecks earlier in the system design, which results in best possible design in fewer design cycles. Our model offers competitive advantages in guiding developers while designing multi-tasking applications in platform based design flow. Experimental results with JPEG encoder design shows that our RTOS model is scalable to complex platforms and very accurate while simulating fast. The results are within 7% of actual board measurements even for industry size applications, while simulating faster than real-time. The experimental results demonstrate that the proposed RTOS model can be used reliably and efficiently for fast, early, and accurate estimation in platform based design approach. In the future, we plan to extend our RTOS overhead model to finer grained detail using non-intrusive time stamping methods.

REFERENCES

[1] M.-K. Chung, S. Na, and C.-M. Kyung. System-Level Performance Analysis of Embedded System using Behavioral C/C++ model. In *VLSI-TSA-DAT*, Hsinchu, Taiwan, 2005.

[2] ESE: Embedded Systems Environment. http://www.cecs.uci.edu/~ese.

[3] FastVeri (SystemC-based High-Speed Simulator) Product. http://www.interdesigntech.co.jp/english/fastveri/.

[4] D. D. Gajski, J. Zhu, R. Dömer, A. Gerstlauer, and S. Zhao. *SpecC: Specification Language and Design Methodology*. Kluwer Academic Publishers, 2000.

[5] A. Gerstlauer, H. Yu, and D. D. Gajski. Rtos modeling for system level design. In *Proceedings of the Design, Automation and Test in Europe (DATE) Conference*, Munich, Germany, March 2003.

[6] T. Grötker, S. Liao, G. Martin, and S. Swan. *System Design with SystemC*. Kluwer Academic Publishers, 2002.

[7] Y. Hwang, S. Abdi, and D. Gajski. Cycle-approximate Retargetable Performance Estimation at the Transaction Level. In *DATE*, Munich, Germany, April 2008.

[8] J.-Y. Lee and I.-C. Park. Time Compiled-code Simulation of Embedded Software for Performance Analysis of SOC design. In *DAC*, New Orleans, USA, June 2002.

[9] LLVM(Low Level Virtual Machine) Compiler Infrastructure Project. http://www.llvm.org.

[10] H. Posadas et al. RTOS modeling in SystemC for real-time embedded SW simulation: A POSIX model. 10(4):209–227, Dec. 2005.

[11] CoMet: Virtual System Prototype Technologies. http://www.vastsystems.com/solutions-architecture-systems.html.

[12] L. Yu, S. Abdi, and D. Gajski. Transaction level platform modeling in systemc for multi-processor designs. Technical Report CECS-TR-07-01, January 2007.