Exploring SW Performance Using Preemptive RTOS Models

Gunar Schirner

Electrical and Computer Engineering
Northeastern University, Boston MA
E-mail: `schirner@ece.neu.edu`

Abstract

With increasing SW content of modern SoC designs, modeling of embedded SW has become critical. For one, analyzing software performance early in the system design flow is now paramount to an efficient implementation. Previous work addressed performance modeling with timing annotated functional models and exposed dynamic scheduling effects with behavioral RTOS models. However, such models insufficiently capture preemption as their cooperative decision making is dependent on the timing annotation granularity. In addition to capturing dynamic scheduling, modeling system overhead (e.g. for context switches) becomes essential for guiding developers when deciding the granularity of multitasking applications. In this paper, we introduce two means to improve accuracy of SW performance modeling: a preemptive RTOS model, and the modeling of system overhead. Our experimental results on multimedia applications significant accuracy improvements when analyzing interrupt latency distribution (within 8% for average and 50th percentile), and modeling systems with high system overhead (less than 10% error). Our model extensions provide improved simulation accuracy and therefore better aid the design space exploration.

1 Introduction

Embedded software plays an increasing role allowing a flexible realization of complex features in embedded systems. However, software development cost starts dominating the overall design cost. The productivity gap, traditionally attributed to hardware design, shifts now toward the software domain. To keep pace with the computation potential and complexity of the underlying hardware, the software has to double every 10 months [8]. Given this pressure, a traditional manual software implementation approach is too time consuming, tedious and error prone to meet the shortened time-to-market demands.

One approach to increase design productivity is system-level design, raising the level of abstraction, hiding complexity of low level implementation details, with the goal of a seamless software and hardware co-design. By moving to higher levels of abstraction system-level design reduces the complexity during development, enabling designers to focus on important algorithmic concepts without the burden of low level implementation details. A seamless co-design of hardware and software requires abstract software modeling early in the design flow, as well as efficient synthesis capabilities to automatically realize an implementation from a high-level model.

This article describes software performance modeling, in particular, functional software modeling for performance evaluation during the design space exploration. Top down ESL flows, such as the System-on-Chip Environment (SCE) [7], as well as the Embedded Systems Environment (ESE) [9], enable developing a system specification in an platform-agnostic format. The designer specifies in a separate process the target platform, the application to platform mapping, as well as the system characteristics (task grouping, priority distribution, communication parameters). The ESL flow then generates automatically the abstract model in form of a host compiled Transaction Level Model (TLM). The TLM is then the basis for performance evaluation. In contrast to traditional ISS-based software simulation approaches, a host compiled TLM offers tremendous benefits in simulation performance, thus enables analyzing larger, more complex multi-processor structures.

1.1 Software Performance Estimation

Both HW platform (e.g. processor type, memory hierarchy), and SW platform (e.g. task/data granularity, selection of scheduling policy, priority distribution and the selection of an appropriate RTOS) significantly influence SW performance on the final target platform. A TLM, generated in the ESL flow, should reflect the effects of the above design choices to enable informed decisions during exploration.

To discuss delay contributors, Figure 1 illustrates execution of two tasks. The high priority task T_1 executes in the beginning. When acquiring a non-available semaphore, T_1 starts pending, and the lower priority task T_2 is dispatched. Later at t_4, T_2 releases the semaphore and thus the higher

978-1-4244-7074-7/10/$26.00 ©2010 IEEE
DOI 10.1109/RSP_2010.SS2

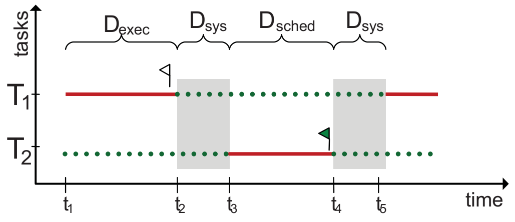


Figure 1. Delay contributors in SW execution

priority T_1 regains execution. Observing this stretch of execution, three coarse delay contributors can be identified:

D_{exec} The execution of straight line code without preemptions or context switches (e.g. t_1 - t_2, t_3 - t_4);

D_{sched} The delay due to dynamic scheduling of other tasks (t_3 - t_4 for T_1)

D_{sys} The system overhead when performing dynamic scheduling (e.g. context switch t_2 - t_3).

Reflecting D_{exec} can be achieved by timing back annotation. The SW model code is instrumented with wait-for-time statements, which emulate target-specific progress of time by advancing simulation time during model execution. Examples for determining retargetable execution time include offline static methods [21, 19, 2], as well as on-line code profiling methods such as [24].

To observe delays due to dynamic scheduling D_{sched}, abstract RTOS models are employed. Approaches have been described in the literature [24, 12, 16, 12]. However, current RTOS models poorly support preemption as the timing annotations representing D_{exec} define the granularity of scheduling decisions. Scheduling decisions are made only at the boundaries of wait-for-time statements, thus the resulting simulation is similar to cooperative multitasking. Accurate emulation of preemption would require fine grained annotation (e.g. at C-statement level) which would slow down simulation speed, and would require fine grained annotation information which may not easily be available for a given application. In this paper, we first show an example of D_{sched} modeling and introduce preemption support without increasing wait-for-time statements.

While D_{exec} and D_{sched} dominate the overall execution, D_{sys} is an important measure for guiding the SW developer in parallelizing a given application in terms of granularity of data parallelism and parallelism in the control flow. A coarse grain parallelization may not sufficiently profit from an underlying multi-core architecture. A too fine grained separation, on the other hand, may unnecessarily increase the number of context switches, thus increase D_{sys} and therefore decrease system performance [4]. Hence, modeling of D_{sys} is important for judging this balance. With current modeling techniques, the negative effects of such a design choice are only discovered when executing on the final system or on an ISS-based model, which would lead to an expensively long design cycle, or slow simulation speed of multi-core systems, respectively. To increase design efficiency, modeling of system overhead, D_{sys}, is required.

In this paper, we present our approach for cycle-approximate TLMs for SW performance evaluation. In particular, the paper focuses on two aspects. Section 2.1 introduces an RTOS model which reflects D_{sched} including preemptions without requiring finer grained timing annotations. Thus, it enables more accurate modeling interrupt latencies in abstract models. Second, Section 2.2 shows an approach to efficiently capture and express system overhead in early SW models and thus guides developers in parallelizing their application. The effectiveness of the approaches are shown using media examples. Section 3.1 examines the system overhead when parallelizing a JPEG encoder application, demonstrating the importance of D_{sys}. Following that, Section 3.2 demonstrates advantages of our preemptable model when analyzing the interrupt latency in a combined JPEG encoder and MP3 decoder application.

1.2 Related Work

System-level modeling has become an important means to improve the SoC design process. System Level Design Languages (SLDLs) for capturing system models have been developed (e.g. SystemC [13], SpecC [11]).

Significant research effort focuses on early performance estimation of execution delay D_{exec}. Estimation techniques [21, 19, 1, 2] analyze the software execution path however abstract processor datapath details. The basic approach is to multiply a statically determined cost for each operation type with the number of their occurrence. In contrast, [22, 5] take the processor datapath structure into account, however utilize ISS-based simulation at the expense of speed. The commercial systems [30, 10, 29] provide fast system simulation models, but have limitations in integrating custom hardware components. To avoid the aforementioned limitation, our RTOS model is using the estimation technique proposed by [17] for application code performance estimation.

Abstract RTOS models on top of SLDLs have been developed to emulate D_{sched}. [6] proposes SoCOS, a high-level RTOS model. It interprets a proprietary language, describing RTOS characteristics, using a specialized simulation engine. Our proposed solution uses a standard unmodified discrete event simulator. [16] describes an RTOS centric cosimulator, using a host compiled RTOS. However, it does not include target execution time simulation. [12, 32] introduces abstract scheduling on top of SpecC and SystemC respectively, providing scheduling primitives found in a typical RTOS and allows modeling of target-specific execution timing. However, both emulate preemption only at the granularity of the timing annotation.

Less support is available for preemptive abstract RTOS models. [28] presents s fixed-priority preemptive multi-

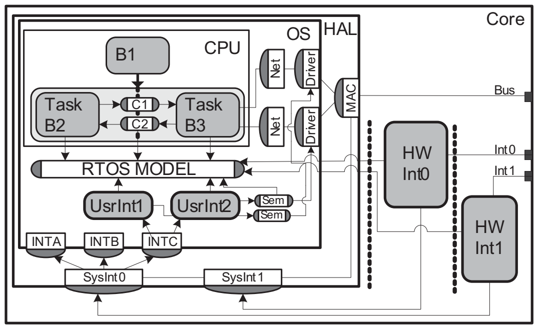


Figure 2. Layered processor model with preemption support

tasking model, however uses SpecC specific concurrency and exception mechanisms and poses strict limits on inter-task communication. [24] presents an abstract RTOS model with a POSIX API on top of SystemC. It combines online estimation of D_{exec} through overloading each basic operator and dynamic delay calculation and a dynamic scheduling model. This combination enables efficient options for handling of timer interrupts. However, the approach suffers in simulation performance due to the online calculation of D_{exec}. [15, 14] employ a prediction of future interrupts. [15], however, uses an unrealistic single thread assumption and [14] relies on additional user input for the prediction.

2 Software Modeling

This section describes modeling of software performance in two aspects. Fist, it introduces preemptive abstract scheduling for an accurate estimation of D_{sched}. The second portion of this section outlines estimation and modeling of the system overhead D_{sys}.

2.1 Modeling of Preemptive Dynamic Scheduling

Figure 2 shows our layered processor model which is largely based on [27]. The model is captured in the SpecC SLDL and natively compiled. Its innermost layer (the CPU layer) contains the user specified behaviors, grouped to tasks. The user behaviors are timing annotated according to the selected target processor yielding D_{exec}. The OS layer surrounding the CPU implements application specific communication layers, for synchronization and communication with external components.

To facilitate dynamic scheduling emulation, each statement in a user behavior that could potentially impact scheduling is wrapped to interact with the abstract RTOS model (examples include: task create, - suspend, - resume, semaphore acquire, - release). The abstract RTOS [12] maintains a task state machine similar to a regular RTOS and dispatches tasks using primitives of the underlying SLDL (e.g. events). It sequentializes the task execution according to the selected scheduling policy.

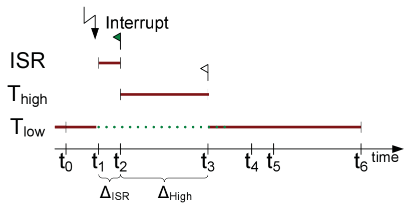
In addition to typical primitives the abstract RTOS provides an interface to emulate time progression. The wait-for-time statement represents execution time: the time needed to execute a set of instructions on the target CPU [17]. Scheduling decisions in typical TLM based RTOS models are made at the boundaries of wait-for-time statements (i.e. at fixed points similar to cooperative multitasking). Interrupt Service Routines (ISRs) are modeled as highest priority tasks, which are suspended at startup and later released by an IRQ for execution.

Figure 3 illustrates how decision granularity influences simulation accuracy. It shows a preemption on a processor, a non-preemptable TLM-based RTOS and on a ROM-based preemptable RTOS model which we will introduce later.

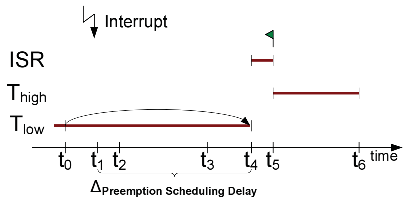
First, Figure 3(a) depicts a preemption on a real processor as a reference. While the low priority task T_{low} executes, an interrupt preempts at t_1 and triggers the ISR. The ISR activates T_{high}, which computes until acquiring a semaphore at t_3. Subsequently, the preempted T_{low} resumes. Figure 3(b) shows preemption in a TLM-based RTOS. The section executed by T_{low} is annotated with a single wait-for-time statement (from t_0 to t_4 – depicted by an arc). Since the TLM-based RTOS evaluates scheduling at boundaries of wait-for-time statements, the interrupt is evaluated only at t_4. Then, it schedules first the ISR, followed by T_{high}. Note the inaccurate timing. T_{high} finishes late at t_6 (instead of t_3). Conversely, T_{low} finishes early at t_4 (instead of t_6).

Using the Result-Oriented Modeling (ROM) approach, we can introduce preemptive modeling [26] to overcome the limitations of the TLM-based RTOS. ROM is a general concept for abstract yet accurate modeling of a process that was demonstrated for communication modeling [25]. ROM aims to rapidly produce the *end result* by hiding and eliminating internal states. It employs an *optimistic prediction* approach to determine the outcome (e.g. termination time and final state) of the process already at the time the process is started. While the predicted time passes, ROM records any *disturbing influence* that may alter the predicted outcome. In the end, it validates the prediction and takes *corrective measures* to ensure accuracy.

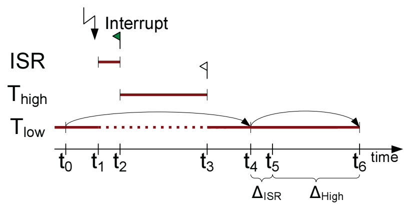
The preemptable ROM-enhanced RTOS differs implementation from a TLM-based RTOS in three crucial elements: (a) integration of interrupts, (b) wait-for-time statements, and (c) dispatch implementation. Interrupt handling has to occur in parallel to executing the modeled tasks, wait-for-time statements have to be preemptable, and the dispatcher has to be expanded to update the preemption record of a preempted task. The most important aspect is the wait-for-time implementation. In the ROM-based version, scheduling decisions are possible *while* wait-for-time



(a) Processor



(b) TLM



(c) ROM

Figure 3. Preemption in priority-based scheduling.

is running. For proper timing, this demands keeping track of the time spend in execution and preemption.

The ROM-based wait-for-time implementation treats the annotated wait time as an initial prediction. The time annotated computation section would execute for this duration if not preempted. This is an optimistic prediction since it is the shortest time this section may finish – a preemption would only add delay. During the progress of time, ROM collects the disturbing influence of preemptions. An external interrupt to a modeled processor can trigger preemption. The interrupt detection logic executes in parallel to all tasks in the processor. Upon detection of an interrupt signal, it uses the ROM scheduler to update the task states and start the preemption chain. The dispatcher of the ROM scheduler updates the preemption record of preempted task(s), noting start and finish of a preemption period. At the end of the any wait period in a wait-for-time, ROM validates the initial prediction. In case of a preemption, the preempted task will have a preemption record in its virtual task control block (TCB). It then updates the wait period reflecting the preemption and waits again.

Returning to the earlier example, Figure 3(c) depicts how ROM handles the preemption. As before, T_{low} starts execution of a new section of code at t_0. Its time progress is simulated by a wait-for-time statement, which ROM uses as an initial prediction. Thus, T_{low} starts waiting until t_4, as indicated by the arc. The interrupt detection detects an interrupt at t_1, and triggers the dispatcher to virtually preempt execution of T_{low}, recording the start time of preemption. The scheduler then dispatches the ISR. Note that although T_{low} still executes the wait-for-time, it is no longer considered RUNNING (but READY).

The ISR activates T_{high} at t_2 and finishes its execution. The scheduler thus dispatches T_{high}, which executes until acquiring a semaphore at t_3. At this time, the scheduler attempts to dispatch T_{low}. Since, however, T_{low} was preempted, the scheduler only updates T_{low}'s preemption record calculating the total preemption duration as $t_{Now} - t_{FirstPreemption} = t_3 - t_1$, which matches $\Delta_{ISR} + \Delta_{high}$. When T_{low} finishes the initial prediction at t_4, it reads the preemption record and waits for the preemption duration until t_6. In contrast to the TLM, with ROM all time stamps match the execution on the actual processor (Figure 3(a)).

The ROM-based RTOS model efficiently implements a preemptable model independent of timing annotations granularity. It uses the D_{exec} as an initial prediction, and treats preemptions and subsequent higher priority task releases as a *disturbing influence*. Additional preemption scenarios and a detailed description can be found in [26].

2.2 System Overhead Modeling

Employing an abstract RTOS model is essential for early SW performance estimation. With the increasing software content in embedded systems and increase of complexity, the importance of abstract OS models will increase. In conjunction with the wider use of multi-threaded applications, modeling of system overheads gains in importance. For the purpose this article, system overhead is considered to contain for example the context switch delay, delay due to inter task communication as well as interrupt preemption.

Reflecting the system overhead D_{sys} in a software simulation is an essential guideline for the developer when partitioning code. In a balanced system, the system overhead is less than 10% of the execution time. However, in a system with a too fine data and/or control granularity (i.e. too many, too small tasks), the system overhead can consume a significant portion of computation time due to frequent context switches. In current models, the system overhead D_{sys} is not reflected and using D_{exec} plus D_{dyn} is an insufficient indicator for the final performance as they do not expose the potential bottleneck of RTOS overheads.

Modeling RTOS overhead is challenging as the actual overhead depends on RTOS, CPU, CPU configuration (e.g. caching), and processor state. Further complications are posed a limited source code availability especially for a commercial RTOS, as well as structural/organizational differences between implementations. Therefore, a static code analysis similar to D_{exec} is not feasible.

Instead of using a static analysis approach, we developed a profiling application [18] with a deterministic scheduling sequence given a particular scheduling policy. It invokes RTOS primitives in a known sequence and captures time stamps from an external timer. We use the profiling information to analyze RTOS overheads on the RTOS API level without source code analysis. We characterize each RTOS

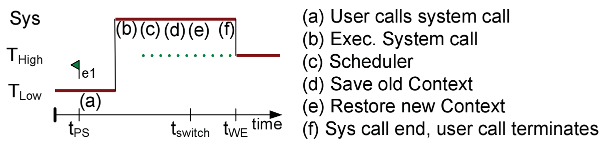


Figure 4. sem_give() with context switch

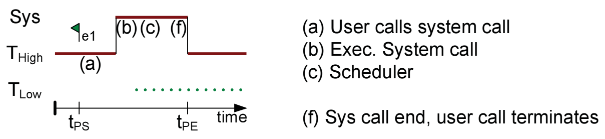


Figure 5. sem_give() without context switch

on the supported configuration(s), and record the overhead model in a database. Once a RTOS is characterized, its overhead model can be used in any user application simulation and thus to many designs. Figure 4 and Figure 5 show a small excerpt of the profiling application demonstrate releasing a semaphore which does, or does not result in a context switch.

Figure 4 illustrates the case when releasing a semaphore results in a context switch to a higher priority task. Two tasks T_{high} and T_{low} acquire and release a shared semaphore. The profiling application records time stamp t_{PS} when task T_{low} releases the semaphore. As a result T_{high} becomes ready, since it earlier acquired the previously non-available semaphore. In the context of the higher priority T_{high}, it records t_{WE} after the corresponding *sem_take* has terminated.

In Figure 5 the roles are switched between T_{high} and T_{low}. Now, T_{high} releases the semaphore on which T_{low} waiting. Due to the lower priority of T_{low}, no context switch is performed. Both time stamps T_{PS} and T_{PE} are taken in the context of the high priority task before and after terminating the system call.

By correlating the time stamps in both measurements, the execution time for *sem_give()*, as well as a rough estimation of the context switch time can be made:

$$\begin{aligned} Dur(sem\_give) &= t_{PE} - t_{PS} \\ Dur(switch) &= t_{WE} - t_{PS} - Dur(sem\_give) \end{aligned}$$

Similarly, we analyze other synchronization and communication primitives. We use the time stamp information to attribute delays to individual components of D_{sys}. The delay information of all primitives is then stored as a RTOS delay model in our database.

During instantiation in a particular model instance, the selected RTOS' characteristics are retrieved and the RTOS model is configured to reflect the selection. Upon execution of an RTOS primitive, the according delay is induced. Note, we use a simplified stateless delay model in favor of simulation speed. It abstracts away many influences (e.g. number of total, waiting, and manipulated tasks, scheduler implementation) and therefore our model is not cycle-accurate. Nonetheless, it yields valuable feedback for estimating system performance and guiding application developers.

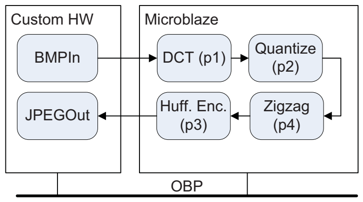


Figure 6. JPEG Encoder.

3 Experimental Results

In order to evaluate the efficiency of the proposed approach, we apply it to two media applications. We first evaluate the system overhead modeling in context of a JPEG encoder application. Second, we evaluate the preemptive model with a combined JPEG encoder and MP3 decoder.

3.1 System Overhead

To demonstrate the system overhead modeling, we use a JPEG encoder which we separated into four tasks: DCT, quantization, zigzag and huffman encoding. As Figure 6 illustrates, these 4 tasks are executed on a Microblaze with 100MHz and scheduled by Xilinx's Xilkernel. The processor is assisted by custom HW block for input (*BMPin*) and output (*JPEGout*).

Due to fine task granularity, a high system overhead is expected as each of the 4 JPEG tasks is context switched for each data packet. We varied the input data granularity between 64byte packets down to packets of 8bytes for further analysis. Figure 7 compares the execution times predicted by a TLM without and a TLM with system overhead modeling. As a reference, it also shows the actual execution time, when executing the target platform (Xilinx FF896).

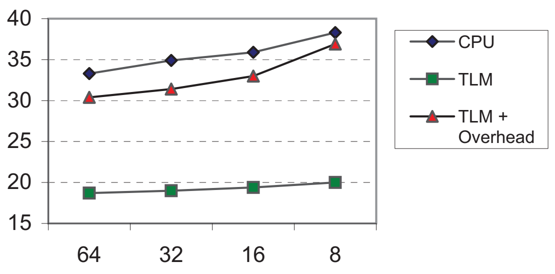


Figure 7. Execution time [Mcycles] over data granularity [bytes].

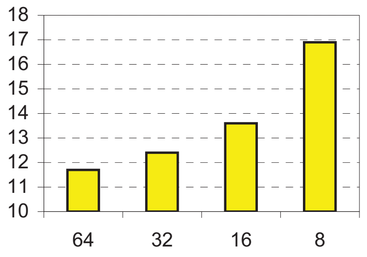


Figure 8. System overhead [Mcycles] estimated by timed RTOS.

Figure 7 indicates the limited expressiveness of the TLM about system overhead. Regardless of the data granularity, it estimates an almost constant execution time of about 19Mcycles. It only shows a small variation with granularity. In total the TLM without D_{sys} model yields highly inaccurate results (46% on average), and does not guide the developer during application parallelization.

The TLM with overhead modeling (D_{sys}) on the other hand, tracks the execution delay on the processor within 10% on average. The estimated execution time significantly increases when using finer grained data blocks. Focusing on the estimated system overhead, as shown in Figure 8, the developer is alerted about the too fine grained data handling. The system overhead increases significantly up to 17MCycles with the fine grained data of 8 byte blocks.

3.2 Interrupt Latency

A multimedia application combining JPEG encoding and MP3 decoding is used for demonstrating the benefits of an preemptive RTOS model. We have implemented it using SCE [3] based on the SpecC SLDL. In our target platform, shown in Figure 9, an ARM7TDMI running μC/OS-II [20] concurrently decodes a MP3 stream and encodes a JPEG picture. Three HW accelerators assist the processor, and an additional set of HW units perform input and output.

We observed in earlier models that a TLM-based RTOS model with function level annotations was *not sufficient*. Unrealistically, the AC97 FIFO frequently ran empty, since the ISR violates its deadline.

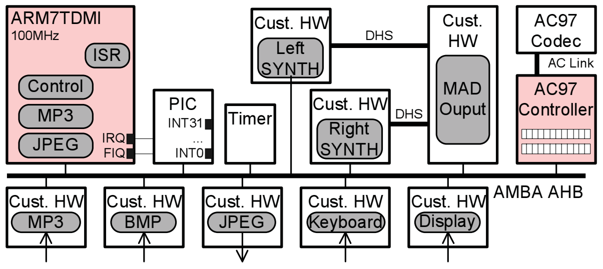


Figure 9. MP3 JPEG media example.

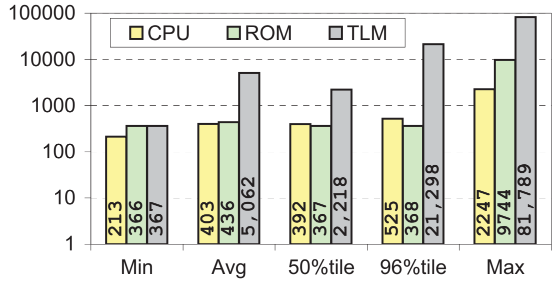


Figure 10. Interrupt latency distribution.

We therefore analyze interrupt latency. In particular, we look at the interrupt to refill the audio output queue in AC97 controller. For our measurements, we define the ISR latency as the time from interrupt release by the AC97 controller until execution of the first instruction in the user ISR. We compare the two abstract models: TLM (without preemption) and ROM (with preemption) against a cycle accurate ISS-based simulation. Figure 10 illustrates the characteristics of the modeled interrupt latency (on a logarithmic scale) during the course of execution.

The measurements document that the coarse grain modeling of preemptive scheduling by the TLM yields a highly inaccurate interrupt latency. The 50^{th} percentile is more than 5 times longer than the actual. The 96^{th} percentile exceeds the actual value by 40x[1]. Hence, a non-preemptable RTOS model is not suited for application that depend on interrupt response time.

By introducing preemptive modeling in the ROM-based model, the latency distribution matches the CPU within 8% in terms of average and 50^{th} percentile. The ROM-based model is therefore a good latency indicator. However, ROM produces very tight latency distribution with minimal latency and 96^{th} percentile being only 2 cycles apart. At this point, we do not model RTOS critical sections, hence ROM does not show the same variation the CPU does.

4 Conclusion

In this paper, we have presented efficient means software performance modeling to guide the design space exploration. In particular, we focused on two aspects.

First, we have introduced a preemptive RTOS model, based on the Result Oriented Modeling principle, which models preemptions independent of the timing annotation granularity. Analyzing interrupt latency in a multimedia application demonstrated tremendous improvements in accuracy. The ROM-based model showed interrupt latencies within 8% for average and 50^{th} percentile when comparing with a real execution, versus a 40x longer 50^{th} percentile in the non-preemptive model.

[1]Note results vary largely by application and annotation scheme[26].

Second, we have expanded the modeling of dynamic scheduling by incorporating RTOS overheads. Our stateless delay model is constructed by characterizing supported platforms with a profiling application, and back annotating a behavioral RTOS model. Our results show the value of overhead modeling in guiding developers when parallelizing applications. Even under high system overhead situations, our model performed well with less than 10% error.

In future, we plan to expand the ROM RTOS to include the influence of critical sections, and investigate stateful RTOS overhead modeling.

Acknowledgments. The majority of the work presented in this paper has been performed while being at the University of California, Irvine. The author thanks for the ESE and SCE development teams of the Center for Embedded Computer Systems for their excellent support. The author thanks Rainer Dömer for is ongoing patient support and creative discussions, Andreas Gerstlauer for the initial abstract RTOS model, Samar Abdi for his work on performance estimation, and Yonghyun Hwang for is implementation of the overhead model.

References

[1] J. R. Bammi, W. Kruijtzer, and L. Lavagno. Software Performance Estimatioin Strategies in a System-Level Design Tool. In *CODES*, San Diego, USA, 2000.

[2] L. Cai, A. Gerstlauer, and D. Gajski. Retargetable Profiling for Rapid, Early System-Level Design Space Exploration . In *DATE*, San Diego, USA, June 2004.

[3] CECS, UC Irvine. SoC Environment (SCE). `http://www.cecs.uci.edu/~cad/sce.html`.

[4] Y. Cho, N.-E. Zergainoh, K. Choi, and A. A. Jerraya. Low Runtime-Overhead Software Synthesis for Communicating Concurrent Processes. In *RSP*, Porto Alegre, Brazil, 2007.

[5] 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.

[6] D. Desmet, D. Verkest, and H. D. Man. Operating system based software generation for system-on-chip. In *DAC*, Los Angeles, CA, June 2000.

[7] R. Dömer, A. Gerstlauer, J. Peng, D. Shin, L. Cai, H. Yu, S. Abdi, and D. Gajski. System-on-Chip Environment: A SpecC-based framework for heterogeneous MPSoC design. *EURASIP Journal on Embedded Systems*, 2008.

[8] W. Ecker, W. Müller, and R. Dömer. Hardware dependent software: Introduction and overview. In W. Ecker, W. Müller, and R. Dömer, editors, *Hardware Dependent Software: Principles and Practice*. 2009.

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

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

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

[12] A. Gerstlauer, H. Yu, and D. Gajski. RTOS Modeling for System Level Design. In *DATE*, Munich, March 2003.

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

[14] Z. He, A. Mok, and C. Peng. Timed RTOS Modeling for Embedded System Design. In *RTAS*, San Francisco, 2005.

[15] K. Hines and G. Borriello. A Geographically Distributed Framework for Embedded System Design and Validation. In *DAC*, San Francisco, CA, June 1998.

[16] S. Honda et al. RTOS-Centric Hardware/Software Cosimulator for Embedded System Design. In *CODES+ISSS*, Stockholm, Sweden, Sept. 2004.

[17] Y. Hwang, S. Abdi, and D. Gajski. Cycle Approximate Retargettable Performance Estimation at the Transaction Level. In *DATE*, Munich, Germany, Mar. 2008.

[18] Y. Hwang, G. Schirner, and S. Abdi. Automatic generation of cycle-approximate tlms with timed rtos model support. In *Proceedings of the International Embedded Systems Symposium (IESS)*, Langenhagen, Germany, 2009.

[19] T. Kempf, K. Karuri, S. Wallentowitz, G. Ascheid, R. Leupers, and H. Meyr. A SW Performance Estimation Framework for Early System-Level-Design using Fine-grained Instrumentation. In *DATE*, Munich, Germany, March 2006.

[20] J. J. Labrosse. *MicroC/OS-II: The Real-Time Kernel*. CMP Books, 2002.

[21] M. Lajolo, M. Lazarescu, and A. Sangiovanni-Vincentelli. A Compilation-based Software Estimation Scheme for Hardware/Software Co-simulation. In *CODES*, Rome, 1999.

[22] 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.

[23] J. Madsen et al. Abstract RTOS modeling for multiprocessor system-on-chip. In *In Proceedings of International Symposium on System-on-Chip*, Tampere, Finland, Nov. 2003.

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

[25] G. Schirner and R. Dömer. Result Oriented Modeling a Novel Technique for Fast and Accurate TLM. *IEEE TCAD*, 26(9):1688–1699, Sept. 2007.

[26] G. Schirner and R. Dömer. Introducing Preemptive Scheduling in Abstract RTOS Models using Result Oriented Modeling. In *DATE*, Munich, Germany, Mar. 2008.

[27] G. Schirner, A. Gerstlauer, and R. Dömer. Fast and accurate processor models for efficient MPSoC design. *ACM Tran sactions on Design Automation of Electronic Systems*, to appear, 2010.

[28] H. Tomiyama et al. Modeling fixed-priority preemptive multi-task systems in SpecC. In *SASIMI*, Nara, Oct. 2001.

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

[30] Xilinx. *Embedded System Tools Reference Manual*. 2005.

[31] S. Yoo, G. Nicolescu, L. Gauthier, and A. Jerraya. Automatic Generation of Fast Timed Simulation Models for Operating Systems in SoC Design. In *DATE*, Paris, March 2002.

[32] H. Zabel, W. Müller, and A. Gerstlauer. Accurate RTOS modeling and analysis with SystemC. In W. Ecker, W. Müller, and R. Dömer, editors, *Hardware Dependent Software: Principles and Practice*. 2009.