Introducing Preemptive Scheduling in Abstract RTOS Models using Result Oriented Modeling

Gunar Schirner, Rainer Dömer

Center of Embedded Computer Systems, University of California Irvine
E-mail: {hschirne, doemer}@uci.edu

Abstract

With the increasing SW content of modern SoC designs, modeling and development of Hardware Dependent Software (HDS) become critical. Previous work addressed this by introducing abstract RTOS modeling [6], which exposes dynamic scheduling effects early in the system design flow. However, such models insufficiently capture preemption. In particular, the accuracy of preemption depends on the granularity of the timing annotation. For an accurately modeled interrupt response time, very fine-grained timing annotation is necessary, which contradicts the RTOS abstraction idea and is detrimental to simulation performance.

In this paper, we eliminate the granularity dependency by applying the Result Oriented Modeling (ROM) technique previously used only for communication modeling. Our ROM approach allows precise preemptive scheduling, while retaining all the benefits of abstract RTOS modeling. Our experimental results demonstrate tremendous improvements. While the traditional model simulated an interrupt response time with a severe inaccuracy (12x longer in average and 40x longer for 96^{th} percentile), our ROM-based model was accurate within 8% (average and 50^{th} percentile) using identical timing annotations.

1. Introduction

Current research work has addressed the increasing software content in modern MPSoC designs by utilizing software generation and abstract modeling of software. Abstract RTOS and processor models have been proposed [6]. They expose the effects of dynamic scheduling on a software processor already in early phases of the design. They have deemed crucial for design space exploration, e.g. for task distribution and priority assignment.

However, current RTOS models poorly support preemption. An RTOS model executing in a discrete event simulation environment uses timing annotation to emulate target specific time progress (i.e. via wait-for-time statements). Scheduling decisions are made at the boundaries of these wait-for-time statements, very similar to cooperative multitasking. Hence, the accuracy of preemption depends on the granularity of the timing annotations (Figure 1).

A real CPU provides the finest granularity, checking at each clock cycle for incoming interrupts. Abstract models can annotate each C-instruction, basic block, function, or coarsely grained each task. However, accurate emulation of preemption requires fine grained annotation (e.g. at C-statement level). On the other hand, using fine grained annotation has two drawbacks. It (a) slows down simulation speed, and (b) fine grained annotation information may not easily be available for a given application.

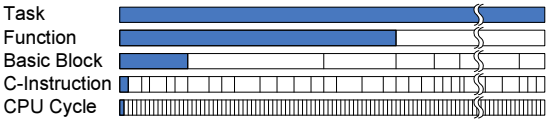


Figure 1. Granularity of timing annotation.

2. Problem Definition

In current modeling of abstract software execution (abstract RTOS on an abstract processor), preemption modeling highly depends on the timing annotation granularity. Scheduling decisions are made at the boundaries of wait-for-time statements. Hence, preemptive scheduling in an abstract model (e.g. after an interrupt) may be delayed by up to the longest time annotation in the whole application. Minimizing this error by using finer grained timing annotation, however, is undesirable due to a slower simulation with the dramatically increased number of wait-for-time statements, and the difficulty to obtain accurate fine-grained timing information. Therefore, preemption is inaccurately emulated in TLM, resulting in intolerable errors e.g. when simulating interrupt response times.

In this paper, we introduce (simulation of) preemption in an abstract model and consequently improve dramatically the accuracy of the interrupt response time without increasing the number of wait-for-time statements.

3. Related Work

Abstract RTOS models have been developed that execute on top of System Level Design Languages (SLDLs) (e.g. SystemC [7], SpecC [4]). [3] proposes SoCOS, a high-level RTOS model. It interprets a proprietary language, describing RTOS characteristics, using a specialized simulation en-

978-3-9810801-3-1/DATE08 © 2008 EDAA

gine. Our proposed solution uses a standard unmodified discrete event simulator. [17] presents modeling of fixed-priority preemptive multi-tasking systems. However, it uses SpecC specific concurrency and exception mechanisms and is limited in inter-task communication. In contrast, our proposed solution, does not rely on SpecC specific primitives and provides full inter-task communication.

[6] introduces abstract scheduling on top of SpecC, providing scheduling primitives found in a typical RTOS and allows modeling of target-specific execution timing. However, it emulates preemption only at the granularity of the timing annotation. In this paper, we will eliminate this restriction. [10] describes an RTOS centric cosimulator, using a host compiled RTOS. However, it does not include target execution time simulation.

[15] presents an abstract RTOS model with a POSIX API on top of SystemC. It uses a comparable approach, but differs in several aspects. First, it overloads each basic operator to dynamically estimate target execution timing. This introduces overhead and reduces performance. ROM, in contrast, statically determines target timing, avoiding that overhead. Second, the approach of interrupt handling differs. In case of an unexpected interrupt, [15] splits the current wait statement and immediately queues up a new wait statement with the remaining time. ROM, in contrast, collects all interrupts during the wait statement, and then issues a new one, combining all occurred preemptions. Hence, a performance advantage can be expected under high interrupt load. Properly synchronized access to global variables is supported in both approaches. [15] also provides special handling for unprotected global variables.

[9, 8] provide interrupt modeling by predicting future interrupts. [9], however, uses a simplistic single thread assumption and [8] relies on additional user input for the prediction. Conversely, ROM exhibits neither limitation.

Previously, ROM was successfully introduced modeling communication [16]. We now apply and extend it to model software execution, improving preemption modeling.

4. Abstract RTOS Modeling

We will first describe a current approach of abstract RTOS modeling [6] (subsequently called TLM-based RTOS) and reveal the limitations in preemption modeling. Second, we will introduce the novel ROM-based abstract RTOS and show how it overcomes the TLM limitations.

4.1. TLM-based Abstract RTOS

The TLM-based abstract RTOS maintains a task state machine for each module/behavior as shown simplified in Figure 2. Each action, which potentially changes scheduling, is wrapped to interact with the abstract RTOS model (e.g. task create, - suspend, - resume, semaphore acquire, - release). For example, if a running task starts pending on a non-available semaphore, its state changes from RUN to WAIT (as in a regular RTOS). The abstract RTOS [6], keeps track of all task states 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 RTOS primitives, an 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 [11]. Scheduling decisions 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.

A preemption, as a result of an external interrupt, can occur at any point in time. Since a wait-for-time increases simulation time, a preemption will occur while executing this statement. With the scheduling decision being made only at the end of the time increase, the preemption (dispatch of the selected ISR task) takes effect *after* the wait-for-time statement. This delays preemption scheduling and subsequently increases the latency for an ISR. Figure 3 shows a preemption situation handled by different approaches. We use line styles to indicate task states: a solid line represents RUNNING, dashed line READY and no line indicates the WAIT state. The empty flag indicates pending on a semaphore (or event), a filled flag its release.

First, Figure 3(a) depicts 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} at t_2 and finishes. T_{high} computes until t_3 when acquiring a semaphore. Subsequently, the preempted T_{low} resumes and finishes the section of computation at t_6.

Figure 3(b) shows preemption in the 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 occurring at t_2, is evaluated only at t_4. Then, it schedules first the ISR, then T_{high}. Note that the TLM-based RTOS is highly inaccurate. T_{high} finishes late at t_6 (instead of t_3). Analogous, T_{low} finishes early at t_4 (instead of t_6).

Note that the duration for this preemption scheduling delay depends on the granularity of the timing annotation. It is application dependent, and hence does not have a fixed

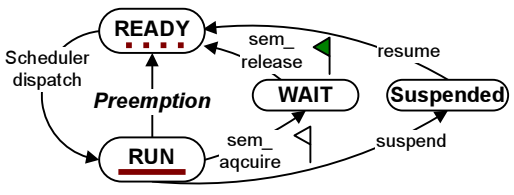
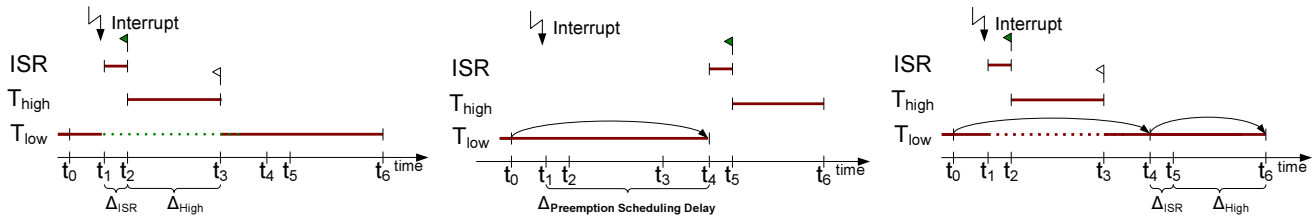


Figure 2. Abstract RTOS task state diagram.



(a) Processor (b) TLM (c) ROM

Figure 3. Interrupt execution in priority scheduling.

bound. This delay can be as long as the longest wait-for-time period in the whole application. To estimate the preemption scheduling delay, we statically analyzed five applications with function level annotations in Section 5. The delay is significant with up to 75,000 CPU cycles. For two applications, 50% of the preemptions will be delayed for at least 5,000 cycles. In this paper, we propose a ROM-enhanced model to remove this inaccuracy.

4.2. Result Oriented Modeling (ROM)

ROM is a general concept for abstract yet accurate modeling of a process that was demonstrated for communication modeling [16]. ROM assumes a limited observability of internal state changes of the modeled process. It is not necessary to show intermediate results of the process to the user, as in a "black box" approach. The only goal of ROM is to produce the *end result* of the process fast. Hiding of intermediate states gives ROM the opportunity for optimization. Often, intermediate states can be entirely eliminated. Instead, ROM utilizes an *optimistic predicts* 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.

4.3. ROM-enhanced Abstract RTOS

Our ROM-enhanced abstract RTOS is based on the same principles as the earlier described TLM-based abstract RTOS. It extends all primitives, which potentially trigger scheduling, to interact with a centralized abstract RTOS model. However, ROM differs in the implementation of three crucial elements: (a) integration of interrupts, (b) wait-for-time statements, and (c) dispatch implementation. As a result, the ROM-based RTOS handles preemptions with higher accuracy by allowing preemption of wait-for-time statements.

The most important aspect is the wait-for-time implementation. While in the TLM-based version, a transition from RUN to READY (Figure 2) was only possible at the end of such a statement, ROM relaxes this assumption. It allows the scheduler to change the task state *while* wait-for-time 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 requested wait time as an initial prediction. This is the duration the process will execute if no preemption occurs while waiting. It is an optimistic prediction since it marks the earliest time this section of computation may finish. During the progress of time, ROM collects the disturbing influence of preemptions. At the end of the wait period, 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.

The dispatcher of the ROM scheduler updates the preemption record of preempted tasks, noting start and finish of a preemption period. 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.

4.3.1 Single Preemption

We will now return to the earlier described example to better explain the interactions within ROM. 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. ROM uses the annotated time as an initial prediction. Thus, the module/behavior of 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}. It sets T_{low}'s state to READY and records the start time of preemption. The scheduler then dispatches the ISR[1]. Note that although T_{low} still executes the wait-for-time, it is no longer in the RUNNING state.

The ISR wakes up T_{high} at t_2 and finishes its execution. The scheduler then dispatches T_{high} as the highest priority READY task. T_{high} accurately executes until acquiring a semaphore at t_3. At this time, the scheduler attempts to dispatch T_{low}. Since T_{low} was preempted, the scheduler updates the preemption record for T_{low} and calculates the

[1]To simplify, we do not show the difference between system and user interrupt handler, which our implementation actually distinguishes.

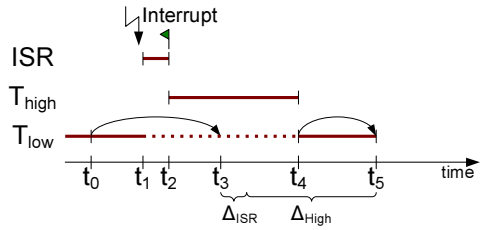


Figure 4. Preemption exceeding prediction.

total preemption duration as $t_{Now} - t_{FirstPreemption} = t_3 - t_1$, which matches $\Delta_{ISR} + \Delta_{high}$. At t_4, T_{low} finishes the initial prediction. It detects the completed 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)).

4.3.2 Preemption Exceeding Initial Prediction

A more complex scenario occurs when the preemption exceeds the initial predicted time, as depicted in Figure 4. Here, the preemption duration (disturbing influence) is not known at the end of the first prediction.

In this example, T_{low}'s initially predicted time for the section extends to t_3. At t_1 an interrupt preempts T_{low}, thus the scheduler records the preemption start and changes T_{low}'s state to READY. At t_3, however, the preemption is not completed. T_{high} still occupies the CPU. Nevertheless, T_{low} wakes up, since the initial predicted time has passed. Here, ROM's wait-for-time detects the incomplete preemption record. It computes the current preemption amount ($t_{Now} - t_{PreemptionStart} = t_3 - t_1$) and clears the preemption record. T_{low} then suspends and waits until being dispatched by the scheduler.

T_{high} finishes execution at t_4. Subsequently, the scheduler dispatches T_{low}, which wakes up and then waits for the previously calculated duration (until t_5). As a result of the preemption, this section of annotated code (i.e. by wait-for-time statement) finishes at t_5. Note that the preemption duration is composed of two portions: (a) the time while T_{low} was waiting (t_1 to t_3), and (b) the time while T_{low} suspends until being dispatched (t_3 to t_4). Only the first portion (a) extends the wait duration.

4.3.3 Cascaded Interrupts.

Besides improving timing accuracy, a ROM-enhanced abstract processor model may also improve accuracy in terms of execution order over a TLM-based RTOS. Again, in the TLM-based version, the decisions are made at the boundary of the wait-for-time. Hence all scheduling inputs (i.e. interrupts) are collected during this time and processed at the end. As a result, a TLM-based RTOS model can show an incorrect execution order on a processor with multiple interrupts (e.g. an ARM7 has two interrupt inputs, an IRQ and higher priority FIQ). Figure 5 shows such a situation,

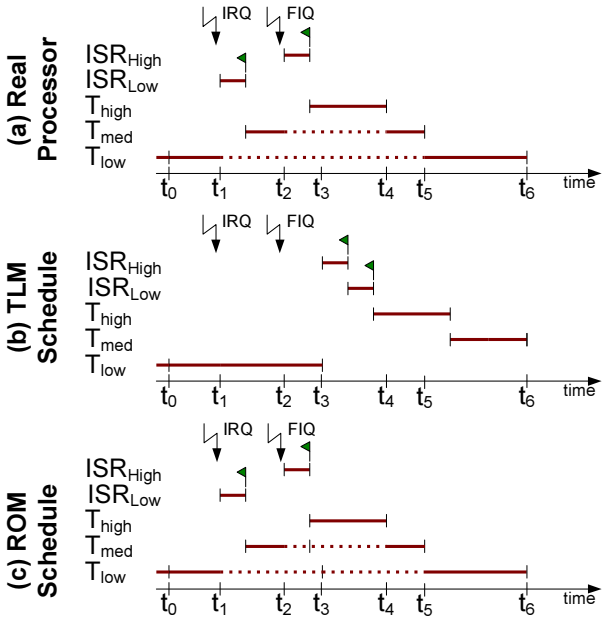


Figure 5. Cascaded interrupts

where two interrupts arrive in a section annotated by one wait-for-time statement.

During an execution segment of T_{low} (from t_0 to t_3), two interrupts arrive: first the lower priority interrupt IRQ at t_1, and second the higher priority FIQ at t_2. On the real processor (a) the corresponding ISRs are executed in sequence (first ISR_{Low} then ISR_{High}). With a traditional TLM (b), however, the execution order is reversed. Since a scheduling decision is done at the end of the wait-for-time statement, the events are accumulated (and cannot be distinguished in their arrival order). As a result, at t_3, the scheduler picks the higher priority FIQ first, creating an incorrect execution order. ROM (c), on the other hand, is able to simulate the correct execution order. Since, it allows a preemption within a wait-for-time statement, the earlier arriving IRQ is properly scheduled first.

5. Analysis of Potential Benefits

In order to quantify the benefits of our ROM-based model, we first statically analyze five applications with function level timing annotations. In a second step, we measure one application.

Our analysis metric is the ISR response time (from signaling the interrupt to start of the ISR). The response time consists of the annotated time and the delay in preemptive scheduling when using a TLM. As stated before, this delay depends on the granularity of the timing annotation. When an interrupt is triggered, the preemptive scheduling delay is equal to the remaining time of the currently running wait-for-time statement. The probability of incurring a delay of T_{del} is the number of all wait-for-time invocations with a delay of at least T_{del} over the total execution duration.

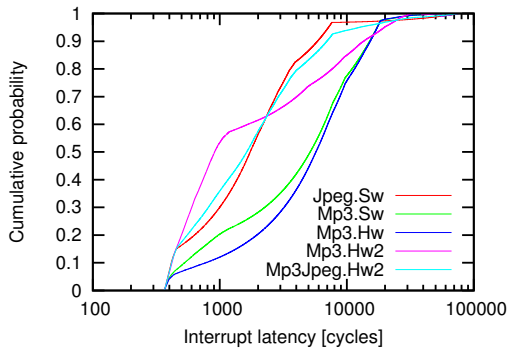


Figure 6. Statically analyzed TLM ISR latency.

More formally, the execution duration (busy time) of an application is captured in N wait-for-time statements, each annotates a duration of W_i and is executed C_i times. With this definition, the application execution time can be computed as $T_{exec} = \sum_{i=1}^{N} C_i * W_i$. The probability $P(T_{del})$ of incurring a delay of T_{del} is then[2]:

$$P(T_{del}) = \frac{\sum_{i=1}^{N} C_i D_i(T_{del})}{T_{exec}}, \quad (1)$$

$$\text{where} \quad D_i(T_{del}) = \begin{cases} 1 & : \quad W_i \geq T_{del} \\ 0 & : \quad W_i < T_{del} \end{cases}$$

Figure 6 depicts the cumulative probability of an ISR latency for a busy CPU. The logarithmic x-axis denotes latency in CPU cycles. Table 1 shows the numeric results.

Table 1. Statically analyzed TLM ISR latency.

	Specified	50%tile	96%tile	Max
Jpeg.Sw	366	1734	7434	75693
Mp3.Sw	366	5146	17750	55190
Mp3.Hw	366	5710	17810	55255
Mp3.Hw2	366	910	21538	38911
Mp3Jpeg.Hw2	366	1629	16291	75932

The preemption scheduling delay, thus the *error* in ISR latency, has a significant spread in a TLM. For our analyzed applications, 50% of ISRs will be delayed by up to 5,710 cycles. For the Mp3.Hw example 46% of ISRs will be delayed between 5,710 and 17,810 CPU cycles. In comparison to the specified delay of 366 cycles, our analysis indicates a potential improvement in the order of two magnitudes. In general, the actual improvement will depend on the application and its timing granularity.

6. Experimental Results

In order to demonstrate the benefits of our ROM-based abstract RTOS model on a real-world design example (Figure 7), we have implemented it on top of SCE [1] using the SpecC SLDL. We realized the ROM-based RTOS without any change in the simulation engine, it only uses standard

[2]For simplification, we assume that the CPU is busy at the time of interrupt and that no other interrupt is currently running.

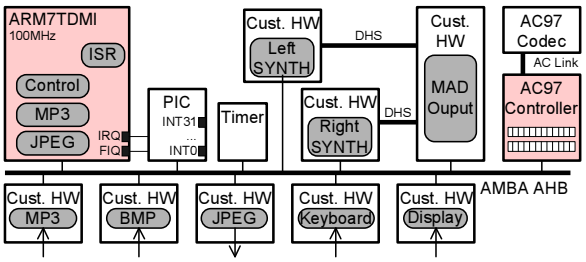


Figure 7. MP3 JPEG media example.

primitives (events and wait-for-time statements). Note that the ROM concept is generic and can be directly applied to other SLDLs such as SystemC as well.

To measure the improvements, we use the ROM-based abstract RTOS in an industrial sized example as outlined in Figure 7. An ARM7TDMI running μC/OS-II [13] concurrently decodes a MP3 stream and encodes a JPEG picture. The processor is assisted by 3 HW accelerators, an additional set of HW units perform input and output. We focus in this simulation on the audio output. The ARM writes the decoded samples into the AC97 controller, which feeds them via an AC-Link to an AC97 codec [12]. Upon a half-filled FIFO, the AC97 controller triggers an interrupt to the ARM which then writes additional samples into the FIFO.

Using a TLM-based RTOS model with function level annotations was *not sufficient* for this example. The AC97 FIFO frequently ran empty, since the ISR violates its deadline. Therefore, we analyzed the interrupt latency for the ISR filling the FIFO. For this paper, we define the ISR latency as the time period from which the AC97 controller releases the interrupt until the execution of the first instruction in the user ISR[3].

Figure 8 shows the ISR latency for three solutions: execution on a cycle accurate ISS [2], simulation using the TLM-based RTOS, and using our ROM-based solution. The logarithmic x-axis denotes the ISR latency in CPU cycles. The y-axis denotes the cumulative probability. As an example, the TLM line reads 0.41 at 1000 cycles, indicating that 41% of the ISR invocations will be delayed by 1000 cycles or less. Table 2 shows the same data in numerical form.

When executing on the ISS, 50% of invocations will only be delayed by up to 392 cycles. The latency has a tight distribution. It has some variance due to critical sections disabling interrupts, and the current instruction's cycle length.

The measurements document that the TLM produces a highly inaccurate interrupt latency. 50% of invocations are delayed by up to 2,218 cycles (5 times higher than actual). The average latency is with 5,062 cycles over 12 times longer than observed on the CPU. The 96^{th} percentile is 40x larger. Hence, the TLM is not suited to simulate an application that depends on the interrupt response time.

[3]Please note, we use a programmable interrupt controller. Hence, the system interrupt handler will first determine the interrupt source, and then invoke the user interrupt handler, which we include in the latency.

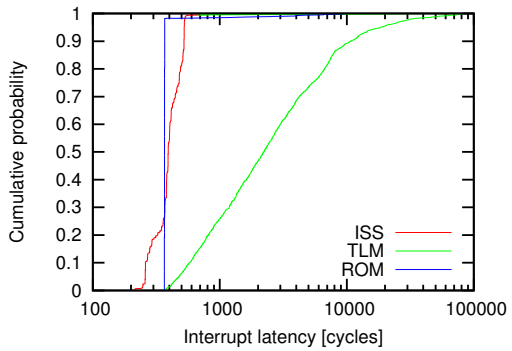


Figure 8. Measured interrupt latency.

Our ROM-based abstract RTOS model, on the other hand, shows a very tight distribution. The minimal latency and the 96^{th} percentile are only 2 cycles apart. Interestingly, the maximum observed latency reached almost 10,000 cycles. Here, the interrupt occurred while the CPU just started another ISR. Since both interrupts use the same priority level, no preemption occurred and the measured ISR started late. The ROM ISR latency distribution matches the CPU within 8% in terms of average and 50^{th} percentile. At this point, we do not model RTOS critical sections, hence ROM does not show the same variation the CPU does.

Table 2. Measured interrupt latency.

	Min	Avg	50%tile	96%tile	Max
CPU	213	403	392	525	2247
TLM	367	5062	2218	21298	81789
ROM	366	436	367	368	9744

Moreover, ROM achieves these improvements using identical timing annotations as used for the TLM. In addition, simulation performance is not affected. The TLM simulated in 22.50 seconds and ROM in 22.56 seconds.

7. Conclusion

In this paper, we have presented a novel approach for modeling preemption in an abstract RTOS model. Our solution is based on the Result Oriented Modeling technique, previously applied only to communication modeling. While a TLM-based RTOS model relies on fine-grained timing annotations to emulate preemptions, our ROM-based model allows accurate preemption at any point. ROM significantly increases the timing accuracy of preemption simulation without demanding fine-grained timing information and without reducing simulation performance.

Our experiments with a real-world example demonstrate tremendous improvements in accuracy. In a TLM, an interrupt response time was on average 12x longer (40x longer for 50^{th} percentile) when comparing to an ISS simulation. ROM, on the other hand, is accurate within 8% for average and 50^{th} percentile. With this accuracy improvement, ROM is an enabler to further expand the use of abstract modeling.

This work is the first to show that the ROM concept is applicable outside of the communication domain. Where in communication modeling it is tied to a particular bus model, here the ROM approach is *not* application specific. Any application scheduled on the ROM-based RTOS will benefit from the enhanced accuracy.

Acknowledgments. We thank Andreas Gerstlauer for providing the initial abstract RTOS. Additionally, we thank the SCE research team for their support.

References

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

[2] M. Dales. *SWARM 0.44 Documentation*. Department of Computer Science, University of Glasgow, Nov. 2000.

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

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

[5] A. Gerstlauer and W. Mueller. OS Modeling. In *HdS Workshop at DAC*, San Diego, CA, Sept. 2007.

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

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

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

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

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

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

[12] Intel Corporation. *Audio Codec '97 Component Specification*, Sept. 2000.

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

[14] 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.

[15] 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.

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

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

[18] 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.