High Level RTOS Scheduler Modeling for a Fast Design Validation

Fabiano Hessel, César Marcon, Tatiana Santos[†]

PPGCC - FACIN – PUCRS - Porto Alegre, RS – Brazil

Fabiano.Hessel@pucrs.br

[†]UNISC - DI - Santa Cruz do Sul, RS – Brazil

tatianas@unisc.br

Abstract

The use of higher level specification models will open new sceneries for optimization and architecture exploration like CPU/RTOS tradeoffs. Scheduling decision for real-time embedded applications has a great impact on system performance and, therefore, it is an important issue in RTOS design. Moreover, it is highly desirable to have the system designer able to evaluate and select the right scheduling policy at high abstraction levels, in order to allow faster exploration of the design space. In this paper, we address this problem by introducing an abstract RTOS scheduling model as well as a new approach to refine an unscheduled high level model to a high level model with RTOS scheduling. This approach is built on the top of the standard SystemC kernel and enables the system designer to quickly evaluate different scheduling policies and make the best choice in early design stages. Furthermore, we present a case study where our model is used to simulate and analyze a telecom system.

1. Introduction

The utilization of embedded processors for real-time embedded applications has been growing rapidly in recent years. 90% of recent System-on-chip (SoC) designs include at least one processor [1]. In this scenario, more and more tasks, traditionally performed by specific but inflexible hardware, will now be executed by software on dedicated programmable processors [2]. However, running multiple tasks on the same processor requires some Operating System (OS) support, in order to perform efficient task scheduling.

Real-Time OS (RTOS) scheduler achieves more deterministic scheduling of tasks where certain priorities need to be followed. This ensures that tasks are scheduled for execution according to their completion deadlines. The scheduling policy used by the RTOS scheduler has a great impact in the overall system performance. Nevertheless, validate different scheduling policies at low abstraction levels remains quite high cost.

Raising the abstraction level is widely used as an interesting alternative to enable faster exploration of the design space at early stages. The correctness of real-time applications is determined by the combination of the computation result and time properties. These aspects make real-time applications different from any other, as it is not possible to analyze them statically at compile time. The actual timing properties need to be checked at run time through target specific code implementation.

In general, although high level simulations are not cycle accurate, it provides enough information to design decisions with a high confidence. Represent the scheduling behavior through a high level model is not trivial due to the lack of information. As a consequence, the timing properties of the system design changes from the high level model to implementation, and the designer has to tune code delays or task priority assignments at a final stage of system design, which is both error prone and time consuming task.

In this paper, we introduce a high level RTOS scheduling model. The main goal is to provide an efficient approach to abstract the dynamic scheduling behavior and adjust the scheduling policy at higher abstraction levels.

This approach is provided while maintaining the standard kernel of SystemC unchanged, by means of a set of specification rules, and a support library built on the top of the SystemC standard library. This is possible thanks to an abstraction technique at transaction level, which can integrate any new scheduling policy that can be abstracted over the underlying simulation kernel.

Transaction Level (TL) is an emergent description level for system level design [3][4]. TL Modeling (TLM) is sufficient to represent the events ordering and efficient for fast high level evaluation. In order to capture the TL dynamic scheduling behavior, we introduce a set of refinement steps to generate a TLM with RTOS scheduling from an unscheduled TLM. This is necessary because using a detailed RTOS is a contra sense, as the system model is highly abstract.

This remaining of this paper is organized as follows: Section 2 presents some related work; Section 3 presents the abstract RTOS model; Section 4 describes the scheduling model refinement; Experimental results with a telecom system that consists of fifty tasks with four priority levels are the subject of Section 5; and finally, Section 6 presents conclusions and future work.

2. Related Work

Kohout [5] describes the Real-time Task Manager as a processor extension that minimizes the drawbacks associated with RTOSs by supporting, in hardware, common RTOS operations that are bottlenecks to system performance. Adomat [6] proposes an exclusive external hardware module designed to perform RTOS functions. This model improves performance, but it does not allow existing RTOSs to easily take advantage of its offerings.

Wang [7] proposes a high level abstract model and synthesizes an operating system based on device drivers. Yi [8] proposes a virtual synchronization technique for a single processor. It runs only application tasks on the Instruction Set Simulator and models the RTOS in the cosimulation backplane.

Cortadella [9] presents an approach to combine static scheduling and dynamic scheduling in software synthesis. Tomiyama [10] describes a technique for modeling fixed-priority preemptive multi-tasking systems. This model does not support different scheduling algorithms, and inter-task communication.

Gauthier [11] proposes a method for automatic generation of application-specific OS and correspondent application software for a given target processor.

Desmet [12] proposes a high level model of a System-on-Chip Operating System. It is used for modeling, simulation and analysis of the system, besides the implementation through gradual refinements. The focus of this work is on task concurrency issues. However, this system requires own proprietary simulation engine and a manual system model creation.

Gonzales [13] presents an abstract RTOS model using master-slave timed SystemC, which allows to model and analyze the behavior of a complex system that has a RTOS application running on a multiprocessor.

Gerstlauer [14] describes an RTOS model, which is effectively a set of commonly used RTOS services, to extend the original ability of SpecC language to handle the interleaved execution behavior of dynamic schedulers. The adaptation of this model to another System Level Design Language may be a hard and complex task, due to lack of support to model common services as preemption and true multitask execution.

The abstract RTOS model proposed here is similar to Gonzales and Gerstlauer approaches. However, our approach is more generic, since it does not limit to master-slave library or a specific SLDL. Our approach is built on the top of the SystemC library, but it can be directly integrated into any system-level design flow. The distinguishing contribution of our approach is that the support is provided without change the standard kernel. Due to this aspect, we have a powerful and flexible high level RTOS model tailored to validation and design space exploration aspects.

3. RTOS Model

SystemC lacks support to model the dynamic real-time behavior commonly found in embedded software. Typically, SystemC does not provide mechanisms to preempt and resume threads during execution time. In order to allow the aforementioned problem, we developed a RTOS model with some language extensions, and a set of specification rules.

The RTOS model is incorporated into the RTOS TL library and can be parameterized in terms of task parameters. This issue is very useful to represent different application classes (e.g. computation-intensive). The library provides RTOS models with different scheduling algorithms. Also, our RTOS model supports both periodic and non-periodic real-time task models. The RTOS model provides two major categories of services: *OS management* and *Task management*. Figure 1 shows the interface of the RTOS model.

```
void sc_rtos_init();
void sc_rtos_reset();
void sc_rtos_task_suspend(id);
void sc_rtos_task_resume(id);
void sc_rtos_task_kill(id);
void sc_task_create(id, priority, period, bcet, wcet, deadline);
void sc_task_notify(id);
void sc_task_end(id);
void sc_task_wait(delay);
void sc_task_end_cycle(id);
```

Figure 1. API of the RTOS model

OS management services are responsible to the initialization of the RTOS. The *sc_rtos_init* initializes the relevant RTOS data structures and starts the multitasking scheduling. The *sc_rtos_reset* reinitializes the RTOS and it is very useful for validation purposes. In order to allow the preemption and resume tasks during execution time, we introduced two primitives: *sc_rtos_task_suspend* and *sc_rtos_task_resume*. These primitives receive the task identification as parameter.

Task management services are responsible to make the interface between the kernel and the system application. The main goal is to provide to the designer an easy way to describe an application class as a set of tasks.

3.1. Task Model

Tasks are modeled to hold all necessary information to execution. Each task is implemented as a PosixThread, allowing preemption and resume by the scheduler. The *sc_task_create* primitive is used to characterize the execution of the task. It defines the task parameters such as identification, priority, period, deadline, Worst-Case Execution Time (WCET), and Best-Case Execution Time (BCET). In addition, this primitive assigns the task to the scheduler, which assigns *idle* status as the initial task state.

Several others standard RTOS primitives are included in the model like task notify (*sc_task_notify*), task termination (*sc_task_end*), and task suspension (*sc_task_wait*). We also introduced the *sc_task_end_cycle* primitive to model periodic tasks. This primitive notifies the scheduler that a task finished its computation in the current cycle.

3.2. Scheduler Model

At high abstraction levels (e.g. system level) we are not interested in the exact task functionality, but rather than that, we need to be able to determine how long it takes to compute the tasks interactions. From this point of view, the first task of the RTOS is to determine which process runs next. The task management, performed by the scheduler, is the most important function in the RTOS model. Our scheduler model assumes that all tasks are independent threads. Each task is characterized by a set of parameters (deadline, period, priority, WCET, and BCET). Moreover, a task may be preempted by a higher priority task.

The task states used in our scheduler model is similar to the classical task states used in the operating system domain, as depicted in Figure 2. Therefore, according to the scheduler model, tasks may be in one of the four basic status of scheduling: *ready*, *executing*, *idle* or *preempted*.

There is at most one task executing at any time. If there is no useful work to be done, just the scheduling task works. As stated before, our model assumes that all tasks are in the *idle state* at the beginning (*sc_task_create*). Each task stays as *idle* while it does not enter in a new execution period, except if the task was preempted. A task goes into the *ready state* when all required data is available or when it enters in a new execution period, remaining *ready* until it is allowed to run. A task goes into the *executing* state when it receives a run command from the scheduler. As stated above, the task will receive this command only when it has all data required, it is ready to run, and the scheduler selects the task as the next one to run. Once the task has finished its computation in the current cycle, it sends a message to the scheduler (*sc_task_end_cycle*) and goes to the *idle state*.

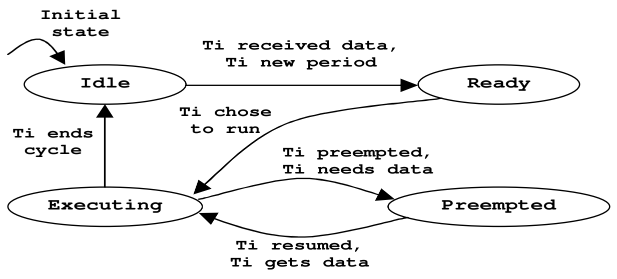

Figure 2. Classical scheduling state of tasks

The task goes to the *preempted state* when it requests a data that is not available (*sc_rtos_task_suspend*). A task can also be preempted by a higher priority one. In both cases, the task will remain preempted until receive a resume command from the scheduler (*sc_task_resume*). When a non-periodic task finished its execution, it sends a terminate message to the scheduler (*sc_task_end*). In this case, the scheduler kills the task (*sc_rtos_task_kill*).

The scheduler is modeled as a SystemC thread process that runs continually. When an executing task goes to *idle* or *ready* states, the scheduler selects a new candidate to run among all *ready* tasks. However, if there is not a candidate task, the scheduler just waits until the next ready task is available. For instance, our scheduler implements First Come First Served (FCFS), Round Robin, Rate Monotonic (RM) and Earliest Deadline First (EDF) scheduling algorithms [15].

4. Scheduling Model Refinement

From the application point of view, the major task of the RTOS is to determine the execution order of the tasks. The scheduler handles this order and it is usually modeled around a priority-based preemptive policy. The input of our model is a non-hierarchical unscheduled model that is refined into a RTOS based multi-task model.

4.1. RTOS-Kernel Model Instantiation

The refinement process starts just after the instantiation of each Processing Element (PE) that composes the abstract architecture. From this one, a RTOS model interface is selected in the RTOS TL library and a run time environment is created for each PE. Hence, the run time environment initializes the internal data structures for the RTOS, and implements the Application Programming Interface (API), which manages the interactions between the abstract application model and the RTOS kernel. After this step, the scheduling refinement environment creates a RTOS main task for each existing task. Those main tasks are the only ones available for the RTOS model to schedule at system start time.

4.2. Task Creation

The main function of the task creation step is to translate all behavior tasks in the application specification model into RTOS-based ones. This is the most important and timing consuming step of the scheduling refinement. Initially, each task behavior inside the PE is checked to see whether it is not a hierarchical task. If such behavior is observed, the task is flatted, and a new one is created. The same approach is used when the task behavior contains parallel processes.

The second step will insert the *sc_task_create* primitive into the task behavior for the creation itself. This primitive activates the task, and assigns *ready* to the scheduler. Finally, the *sc_task_end* or *sc_task_end_cycle* (for periodic tasks) are inserted in the main body of the task. In order to allow preemption and resume by the scheduler, each task is implemented in SystemC as a PosixThread.

4.3. Synchronization Model

The RTOS synchronization model provides services to synchronize concurrent and cooperative tasks, supporting mechanisms that handle inter-processor and intra-processor synchronization problems. Our model implements two main primitives: *sc_task_wait* and *sc_task_notify*.

The *sc_task_wait* primitive makes the current task to wait until another task invokes the *sc_task_notify* primitive or the end of a given time slice. When one of these events happens, the task goes to the idle state. The task is then inserted into a wait task list, becoming disabled for scheduling purposes. The *sc_task_notify* call wakes up a single task that is waiting for data synchronization.

When tasks execute input/output operations, such as send/receive, they need to notify the RTOS scheduler. We implemented this notification by using these same two primitives. An abstract *receive* operation is implemented on lower levels as a receive function aggregated to a *sc_task_wait* call, meaning that the task is waiting for input data. Similarly, an abstract *send* operation is implemented on lower abstraction levels as a send function aggregated to a *sc_task_notify* call. Furthermore, the *sc_task_notify* allows the scheduler to wake up the tasks that are waiting for the sent data.

4.4. Preemption and Synchronizations Refinement

Typically, in high level simulations, *wait* statements are used to model delays, allowing timing advances estimations in those simulations. In our approach, however, the preemption refinement will replace the *wait* statements used to model the delays into the corresponding RTOS calls. Hence, the *sc_task_wait* is used, and it implements a wrapper around the *wait* statement that allows the RTOS kernel to reschedule and to switch tasks.

An important issue is relative to the occurrence of an external interruption. In this event, the execution of a given task can stop, changing the pre-scheduled tasks order. The preemption modeling, in this case, is extremely relevant to assure the accuracy of the model in terms of response time results. Thus, the RTOS kernel uses the *sc_rtos_task_suspend* and *sc_task_resume* primitives to model interrupt preemptions. The accuracy of the preemption results is limited by the granularity of the task delay at the high level models.

Also, in our proposed approach, the synchronization refinement replaces the high level synchronization primitives with RTOS services. This is necessary in order to keep the internal task state of the RTOS model updated. When a given task executes input/output operations, it needs to notify the RTOS scheduler. In this case, an abstract input/output operation wraps the SLDL primitives. The *sc_task_notify* allows the scheduler model to wake up the tasks that are waiting for receive/send data.

5. Case Study

The telecommunication industry has been growing fast in the last few years, especially with the recent development of new technologies such as VOIP (Voice over IP) and wireless devices. Thus, several products already in the market have to be updated in order to aggregate different new features. One of the most popular systems in this market place is the digital Private Branch Exchange (PBX). PBX systems are known as a soft real-time system and therefore an application for using our proposed approach.

The product used in this case study is the model Digistar XT-130, mainly used in commercial environments. The most important issue here is that all system was ad hoc designed to support the real-time requirements. As a consequence, a monolithic system was generated, where application tasks and OS are strongly coupled.

Moreover, it would be necessary to use a RTOS that supports the current as well as the new required features, in order to aggregate the later with few efforts. Systems like this, however, are generally found only when developed with modular designs. A swap between a monolithic to a modular design can imply functionality reduction, mainly in real-time functions. As consequence, more time is required to evaluate the system, reducing the industry profits and possibly resulting market losses.

As alternative, we propose our approach to enable the fast evaluation of different dynamic scheduling policies, allowing the designer to select the optimal scheduler policy at the early design stages.

This PBX is a complex system composed by more than fifty processes, with four priority levels. Around 20% of these processes have real-time requirements. Since the most part of the code is developed in C/C++ and assembly, we proposed a partitioning where system processes are divided as follows: 92% software elements; 6% assembly routines (treated as IP components); and 2% hardware elements. The hardware parts are mapped into Altera FPGA. The software elements are mapped into abstract processors. IP modules and software parts are mapped into AM186ES (AMD 80186) microprocessor and ADSP2185M (Analog Devices) DSP.

For each abstract processor a custom RTOS kernel was generated at the highest abstraction level, using our approach. The abstract RTOS and the system description was refined and targeted to the final architecture.

The abstract channels are refined (communication synthesis step) into shared memory protocol for processors communication, and handshake protocol for FPGA and microprocessor communication. There is no communication between FPGA and DSP processes.

Table 1 depicts the code size (in bytes) achieved for RTOS and the rest of application for both processors.

Table 1. Code size comparison (in bytes)

	AMS186ES	DSP
C/C++	457,976	16,356
Assembly	22,233	27,453
Scheduling Algorithm	7,456	2,489

The PBX model was exercised by testbench vectors extracted from commercial PBX operations during high activity (ten minutes of operation time). The three AM186ES simulations, illustrated in Table 2, show the advantages achieved by high description levels.

Table 2. AMS simulation analysis

Simulation Model	Simulation Time
TL simulation	30 min
RTL simulation	11h 43 min
Cosimulation	185h 5 min

It is clear that the TL model takes only a small fraction of the simulation time, when compared to the RTL and cosimulation ones. One may observe that the TL model does not have the same level of detail as the RTL model and therefore it is not highly accurate. However, global results are always coherent with RTL level simulation. Besides, the time saved using our approach makes it very attractive, especially when the designer needs a fast answer to make a design decision.

Furthermore, we used profile techniques, with the testbench vectors, to estimate the WCET and the BCET of each process. These elements are the entry of each process in TL simulation. Therefore, WCET, BCET, and the execution period replace the process behavior, allowing faster simulation with reasonable accuracy, as RTL simulation confirms. For TL and RTL simulation, the remaining of the system is considered as testbench. On the other hand, the cosimulation considers the joint operation of three simulators (two C/C++ simulators and one VHDL simulator). For these experiments, we used two different scheduling policies: EDF and RM.

Table 3 shows the number of context switches achieved by each scheduling policy as well as the number of RTL constraints fail. The later means the number of times that the real-time processes that did not achieve their deadline in TL simulation and, therefore, must be as low as possible.

Table 3. AMS186ES scheduling analysis

Algorithm	Context switches	RTL constraints fail
EDF	6,315	558
RM	6,280	14,850

The number of context switches is similar in both scheduling algorithms, with less than 1% of difference between EDF and RM. However, EDF algorithm is remarkably better, when comparing the number of RTL constrains fail. In this case, the Earliest Deadline First algorithm generated only 3% of the total number of failings produced by Rate Monotonic.

Therefore, considering context switching, real-time deadline and the low algorithm complexity, we chose EDF as the scheduler policy for AM186ES operation. The majority of DSP tasks are time slices scheduled by a timer interrupt. The total size of AM186ES RTOS is three times larger than the ADSP2185M, due to other additional features, like memory management.

The final delay in the real implementation was higher compared with TL specification. This difference is due to inaccuracies of execution time estimated in the high level model. As discussed before, these inaccuracies are expected in such high level of abstraction and the small amount of time required to development and simulation makes them entirely acceptable. Moreover, when compared to the large complexity required for the implementation of the PBX system, the scheduling refinement environment enables early and efficient evaluation of the dynamic scheduling policies, and enables a fast exploration of the design space.

6. Conclusions and Future Work

This paper addressed the issue of high level RTOS model simulation. We proposed a new approach to quickly evaluate different scheduling policies, providing a way to abstract the dynamic scheduling behavior and adjust each one of them at higher abstraction levels. Moreover, we presented a scheduling environment that refines an unscheduled TLM into TLM with RTOS scheduling.

Our main contribution in the design flow is primarily the automation of the scheduling refinement process that facilitates a fast evaluation of different scheduling policies at high abstraction levels. The environment is written for SystemC, but it can be applied to any C/C++ design flows. Experiments showed the usefulness of this approach in a telecom system design.

Future work includes implementing the RTOS interfaces for commercial real-time operational systems and techniques to handle resource allocation problems.

References

[1] H. Jones. *Analysis of the relationship between EDA Expenditures and Competitive Positioning of IC Vendors for 2003.* http://www.edac.org/resources_profitability.jsp.

[2] W. Wolf. *High-Performance Embedded Computing*. Morgan Kaufman. 2006.

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

[4] L. Cai and D. Gajski. *Transaction Level Modeling: An Overview*. CODES+ISSS, pp. 19-24, 2003.

[5] P. kohout, B. Ganesh and B. Jacob. *Hardware Support for Real-time Operating Systems.* CODES+ISSS, pp. 45-51, 2003.

[6] J. Adomat, J. Furunäs, L. Lindh and J. Stärner. *Real-Time Kernel in Hardware RTU: A step towards deterministic and high performance real-time systems.* 8th Euromicro Workshop on Real-Time Systems, pp. 164-168, 1996.

[7] S. Wang and S. Malik. *Synthesizing Operating System Based Device Drivers in Embedded Systems.* CODES+ISSS, pp. 37-44, 2003.

[8] Y. Yi, D. Kim and S. Ha. *Virtual Synchronization Technique with OS Modeling for Fast and Time-accurate Cosimulation.* CODES+ISSS, pp. 1-6, 2003.

[9] J. Cortadella. *Task generation and compile time scheduling for mixed data-control embedded software.* DAC, 2000.

[10] H. Tomiyama, Y. Cao and K. Murakami. *Modeling fixed-priority preemptive multi-task systems in SpecC.* SASIMI, 2001.

[11] L. Gauthier, S. Yoo and A. Jerraya. *Automatic generation and targeting of application-specific operating systems and embedded system software.* IEEE Transaction on CAD, 2001.

[12] D. Desmet, D. Verkest and H. DeMan. *Operating System based Software Generation for System-on-Chip.* DAC, 2000.

[13] M. Gonzales and J. Madsen. *Abstract RTOS Modeling for Multiprocessor System-on-Chip.* International Symposium on SoC, 2000.

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

[15] A. Silberschatz and P. Galvin. *Operating System Concepts.* John Wiley & Sons Inc, 2000.