Scheduling Refinement in Abstract RTOS Models

FABIANO HESSEL, VITOR M. DA ROSA, and CARLOS EDUARDO REIF
Pontifícia Universidade Católica do Rio Grande do Sul
CÉSAR MARCON
Univeridade Federal do Rio Grande do Sul
and
TATIANA GADELHA SERRA DOS SANTOS
Universidade de Santa Cruz do Sul

Scheduling decision for real-time embedded software applications has a great impact on system performance and, therefore, 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 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 based on SystemC language and enables the system designer to quickly evaluate different dynamic scheduling policies and make the optimal choice in early design stages. Furthermore, we present a case of study where our model is used to simulate and analyze a telecom system.

Categories and Subject Descriptors: C.3 [**Special-Purpose and Application-Based Systems**]: Real-Time and Embedded Systems; B.5.2 [**Register-Transfer-Level Implementation**]: Design Aids—*Design*

General Terms: Design

Additional Key Words and Phrases: Real-time operating systems, transaction level Modeling, RTOS scheduling

1. INTRODUCTION

Raising the abstraction level is widely used as an alternative to enable faster exploration of the design space at early stages, in order to handle the fast-growing complexity of real-time embedded applications.

Authors' Addresses: Fabiano Hessel, Vitor M. Da Rosa, and Carlos Eduardo Reif, Faculdade de Informática, Pontifícia Universidade Católica do RS, Av. Ipiranga 6681, Porto Alegre, RS, Brazil; email: hessel@inf.pucrs.br; César Marcon, Universidade Federal do Rio Grande do Sul, Instituto de Informática, Porto Alegre, RS, Brazil; email: marcon@inf.ufrgs.br; Tatiana Gadelha Serra Dos Santos, Universidade de Santa Cruz do Sul, Departamento de Informática, Santa Cruz do Sul, RS, Brazil; email: tatianas@unisc.br.

The correctness of real-time systems is determined by the combination of the computation result and time properties. These aspects make real-time systems different from any other, as it is not possible to analyze them statically at compile time. A high-level model can determine the computation result correctness at the early design stage of system design. The actual timing properties need to be checked at run-time through target-specific code implementation.

Both the scheduling policy and the system architecture have a great impact in the correctness of these aspects. In general, although high-level simulations are not cycle accurate, it provides enough data to design decisions with a high level of confidence. However, representing the scheduling behavior through a high-level model is not trivial because of the lack of information available at this abstraction level. 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.

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

Transaction level (TL) is an emergent description level for system-level design, which may follow different approaches [Grotker et al. 2002; Cai and Gajski 2003]. In fact, transaction expresses communication exchanges. In other words, transaction informs the relative order of each process communication. Transaction level is an abstraction level, which can lend some focus to the ordering of events. Thus, the TL modeling (TLM) is sufficient to represent the events ordering. The clock abstraction levels and the separation from computation and communication details make the model simpler and efficient for fast high-level evaluation.

In order to capture the dynamic scheduling behavior at a higher level, we propose an approach to abstract the RTOS scheduling. This approach introduces 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 the related work; Section 3 describes how the scheduling refinement process is integrated with the system design flow; Section 4 presents the abstract RTOS model; Section 5 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 6; finally, Section 7 presents conclusions and future work.

2. RELATED WORK

Recently, several works have been focusing on automatic RTOS and code generation. Kohout et al. [2003] describes the real-time task manager (RTM) 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 et al. [1996] 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 and Malik [2003] propose a high-level abstract model and synthesizes an operating system based on device drivers. Yi et al. [2003] propose a virtual synchronization technique to the case where multiple software tasks are executed under the supervision of a RTOS in a single processor. It runs only application tasks on the ISS (instruction-set simulator) and models the RTOS in the cosimulation backplane to achieve faster cosimulation. Cortadella [2000] presents an approach to combine static scheduling and dynamic scheduling in software synthesis. Tomiyama et al. [2001] describe a technique for modeling fixed-priority preemptive multitasking systems based on concurrency and exception-handling mechanisms provide by SpecC. This model does not support different scheduling algorithms, and intertask communication.

Moreover, researchers have realized the importance of dynamic behavior and proposed to include it in system-level design models. Gauthier et al. [2001] propose a methodology for automatic generation of application-specific operating systems and correspondent application software for a given target processor. Desmet et al. [2000] propose a high-level model of a system-on-chip operating system (SoCOS). 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, the SoCOS requires a proprietary simulation engine and a manual system model creation. Gonzales and Madsen [2000] present an abstract RTOS model using master–slave timed SystemC, which allows the modeling and analysis of behavior of a complex system that has a RTOS application running on a multiprocessor. Gerstlauer et al. [2003] describe 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 SLDL language like SystemC may be a hard and complex task, because of a lack of support to model common services as preemption and true multitask execution.

Our abstract RTOS model is similar to the Gonzales and Gerstlauer approaches. However, our approach uses SystemC language, considering TLM specifications [Cai and Gajski 2003]. By introducing extensions in the SystemC scheduler execution model, we have a powerful and flexible RTOS model. Our model can be directly integrated into any SystemC-based system model and design flow.

3. DESIGN FLOW

This section describes an embedded system design flow, starting from a TL specification, which is refined gradually to a hardware and software implementation model, as illustrated in Figure 1. The main issue is demonstrating the design flow for a specific application with automatic generation of an embedded RTOS.

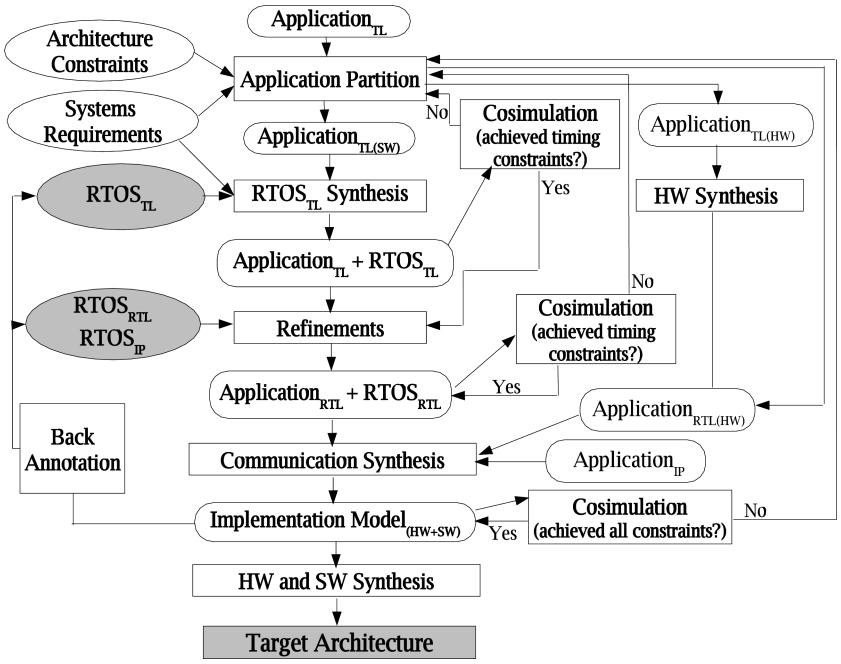

Fig. 1. Design flow.

The system design flow starts with TL specification written in SystemC/C/C++ and IP modules, where the designer specifies the system behavior. The designer informs the system requirements and the architectural constraints, like power consumption limit, real-time constraints, and number of processors of the target architecture. After this, two partitioning steps are accomplished. The first one determines IP components as well as hardware and software processes. Our intention, however, is to propose a new approach for high-level simulations and, therefore, the IP and hardware synthesis are not part of the scope of this work. The intention of this research is to focus in the second partitioning step, where the designer groups the software processes into multiple clusters. Each cluster will be mapped onto a processor in the final implementation. The result is a TLM with several clusters in parallel, each one with a specific behavior. Abstract channels provide the communication between the clusters.

A RTOS TL library was designed to fulfill real-time constraints. This library helps the designer to find the best RTOS scheduling policy at high abstraction levels considering the system requirements, such as performance and power consumption.

Different architecture aspects are also omitted allowing faster design space exploration, mainly regarding scheduling policy for multitasks and multiprocessor.

The RTOS synthesis step introduces the necessary RTOS primitives in all software processes and the scheduling process. These primitives are operating

system calls that allow memory management, interruption request, interprocesses communication, synchronization, and other OS features. At this point, the interprocess communication primitives implement the abstract channels as a device driver.

We use a profiles techniques to estimate the execution time of each process enabling the scheduling mechanisms to preempt software process according to the priority specified by the designer. In addition, a preliminary power estimation of the scheduling mechanisms can be performed. This estimation is based on previous analysis of the scheduling algorithms and the estimation parameter is updated by back-annotation techniques.

As discussed before, the transaction level abstracts part of the communication details, despite the possibility to evaluate the events order and analyze whether all time constraints are achieved with the chosen scheduling mechanism. Thus, TL clearly helps the designer to quickly search for the best scheduling mechanism of each processor and the interprocessor communication mechanism.

After the initial model is complete, the first simulation step is executed. The IP and hardware elements behavior are described as test benches, allowing the validation of the software elements. Once the application achieves all requirements at transaction level, the designer can refine the application description and the selected RTOS for each processor.

The refinement of the application description from TLM to RTL is done manually, generating a synthesizable description. The RTOS refinement is based on two available libraries: one that is the equivalent to RTOS TL at register transfer level and another that is composed by RTOS IP. The RTOS TL refinement to RTOS RTL is quite natural for our design flow, since both represent the same OS at different abstraction levels. In this case, all TL primitives are changed to RTL primitives. On the other hand, the refinement to RTOS IP is harder because of different approaches adopted by IP providers, implying in a few additional manual steps. This level provides more accurate timing and power consumption estimation.

The application and RTOS are validated by simulation and the systems requirements are evaluated. If the constraints are achieved, the flow goes to the next step, otherwise another scheduling policy or hardware/software partition can be chosen and evaluated.

For IP, hardware and software components communication interfaces are synthesized to hardware RTL according to the communication protocol and the target architecture. The design flow supports interprocess communication synthesis with shared memory, rendezvous, FIFO, and buses. Nevertheless, the communication synthesis problem is not addressed in this work. An alternative for this issue is discussed by Dziri et al. [2000].

A hardware/software cosimulation is the last validation step. It uses one simulator for each processor and one simulator for hardware components. This step is usually longer than the others, mainly because of the input simulator vector. However, the cosimulation step is associated with an accurate power model, which allows feedback of the achieved power value to the RTOS libraries, improving any possible next partition.

```
void sc_rtos_init( );
void sc_rtos_reset ( );
void sc_rtos_task_suspend (id);
void sc_rtos_resume (id);
void sc_task_create (id, name, 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);
```

Fig. 2. API of the RTOS model.

As the last step of the design flow, hardware and software are synthesized to the target architecture. Our first approach considers a single FPGA as target architecture. In this context, hardware components are synthesized using specific FPGA commercial tools. For each software cluster, all RTOS primitives are mapped to the correspondent RTOS API, enabling the compilation of the code into the processor instruction set. Each compiled code produces the executable code for each processor.

4. THE RTOS MODEL

As mentioned previously, our RTOS model is implemented based on the SystemC language. However, SystemC lacks support to model the dynamic real-time behavior commonly found in embedded software. Typically, SystemC does not provide a mechanism to preempt and resume a thread during execution time. In order to allow the aforementioned problem, we developed some language extensions.

The RTOS model is incorporated into the RTOS TL library and can be parametrizable in terms of task parameters. The library provides RTOS models with different scheduling algorithms. Our RTOS model also supports both periodic and nonperiodic real-time tasks. The RTOS model provides two major categories of services: OS management and task management.

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. In addition, 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* that preempts a task and *sc_rtos_task_resume* that resumes a task. These primitives receive the task identification as parameter. Figure 2 shows the interface of the RTOS model.

Task management services are responsible to make the interface between the kernel and the system application. The main goal is to provide to the user an easy way to describe an application as a set of tasks. In the following sections we will discuss the task model, scheduler model, synchronization model and, finally, the refinement of the scheduling model.

4.1 Task Model

The tasks are modeled to hold all necessary information to execution. Each task is implemented as a PosixThread, allowing preemption and resume by the

```
1.  int sc_main(int argc, char *argv[]) {
2.    system sys_ex("System example");
3.    ...
4.    sc_start(100000, SC_NS);
5.  }

6.  class system : public sc_rtos {
7.    system(sc_module_name name) : sc_rtos(name)  {
8.      task t1 = new task(id1, "t1", 1, 80, 14, 8, 30);
9.      sc_task_create(t1);
10.     task t2 = new task(id2, "t2", 3, 60, 12, 11, 25);
11.     sc_task_create(t2);
12.     ...
13.     sc_rtos_init(1);
14.   }
15. };

16. class task: public PosixThread, public sc_module {
17.   task(id, "task_name", priority, period, wcet, bcet, deadline) :
18.     sc_module("task_name") {
19.     ...
20.   }
21.   run() {
22.     while(true) {
23.     // task behavior pointed by id
24.     sc_rtos_end_cycle();
25.   }
26. };
```

Fig. 3. Task modeling.

scheduler. The *sc_task_create* primitive is used to characterize the execution of the task. It defines the task parameters, such as identification, name, 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. Figure 3 presents a partial source code example of task modeling. The system *sys_ex* is initialized (line 2) and executed by 100,000 ns (line 4). Moreover, two tasks are modeled in this example: *t1* and *t2* (lines 8 and 10). The task *t1* is created with the following parameters: identification = id1, name = t1, priority = 1, period = 80, WCET = 14, BCET = 8, and deadline = 30. The task *t1* is assigned to the scheduler in line 9. The RTOS scheduler is then initialized with time slice (line 13). All tasks are derived of PosixThread class (line 16).

4.2 Scheduler Model

At the 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

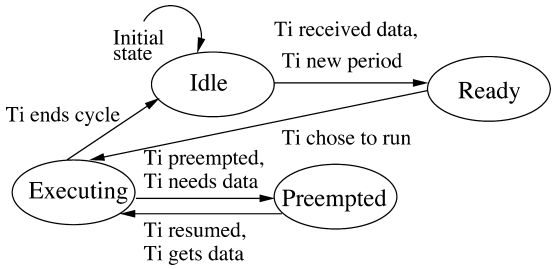

Fig. 4. Scheduling state of tasks.

determine which process runs next, i.e., the RTOS needs to decide the execution order of the tasks. 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. Hierarchical tasks need to be flatted. Each task is characterized by deadline, period, priority, WCET, and BCET. Moreover, a task may be preempted by a higher priority task.

The scheduler model proposed in our work is similar to the one used in Gonzales and Madsen [2001]. Thus, according to the scheduler model, tasks may be in one of the four basic scheduling status: *ready*, *executing*, *idle* or *preempted*, as depicted in Figure 4. 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 period. A task goes into the *ready* state when all required data is available or when it enters in a new 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, 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. However, 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 it receives a resume command from the scheduler (*sc_task_resume*). When a nonperiodic 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 (*sc_thread*) 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. This selection is performed according to scheduler algorithm adopted for that model. However, if there is not a candidate task (*ready* list is empty), 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 Early Deadline First (EDF) scheduling algorithms [Silberschatz and Galvin 2000].

5. 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. In this part of the work, we will use three different scheduling algorithms (Round Robin, RM, and EDF) to illustrate how a RTOS scheduler can be modeled in TLM and how our model can help choosing the scheduling policies at higher abstraction levels. The input of our model is a nonhierarchical unscheduled model that is refined into a RTOS based multitask model. This refinement process is automatically performed by a scheduling refinement environment.

5.1 RTOS-Kernel Model Instantiation

The refinement process starts just after the instantiation of each processing element (PE). As discussed before, these PEs are initially arranged in an unscheduled model. From this one, a RTOS model interface is selected in the RTOS TL library and a run-time environment is created for each PE, taking into account the designer choices regarding the scheduler policy. 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 application and the RTOS kernel. After this step, the scheduling refinement environment creates a RTOS main task for each existing task on the model. Those main tasks are the only ones available for the RTOS model to schedule at system start time, as they wrap around the top-level task behavior.

5.2 Task Creation

The main function of the task creation step is to translate all behavior tasks in the specification into RTOS-based ones. This is the most important and time-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. These primitives allow the RTOS model to control the tasks. In order to allow preemption and resume by the scheduler, each task is implemented as a PosixThread in SystemC.

5.3 Synchronization Model

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

The *sc_task_wait* primitive causes 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.

5.4 Preemption and Synchronizations Refinement

Typically, in high level simulations, *wait* statements are used to model delays, allowing timing advances 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 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.

In our proposed approach, the synchronization refinement also 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.

6. 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), QoS (quality of service), wireless devices and so on. Thus, several products already in the market have to be updated in order to aggregate these 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 [Wolf 2000] and, therefore, an application for using our proposed approach.

The product model used in this case study is the Digitel XT-130, widely used in commercial environments. The main issue is that all system was ad hoc

Table I. Code Size Comparison (in Bytes)

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

Table II. AMS Simulation Analysis

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

designed to support the real-time requirements. Hence, a monolithic system was generated, where application and OS are strongly coupled. Moreover, it would be necessary to use an OS 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 an 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.

The 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 in our design flow as IP components), and 2% hardware elements. The hardware parts are mapped into Altera FLEX-10KE FPGA. The software elements are mapped into PEs, as described in Section 4. IP modules and software parts are mapped into AM186ES (AMD 80186) microprocessor and ADSP2185M (Analog Devices) DSP.

For each 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 I depicts the code size (in bytes) achieved for RTOS and the rest of application for both processors.

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

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

Table III. AMS186ES Scheduling Analysis

Scheduling Algorithm	Context Switches	RTL Constraints Fail
Early Deadline First	6,315	558
Rate Monotonic	6,280	14,850

the TL model does not have the same level of detail as the RTL model and, therefore, 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 test-bench vectors, to estimate the WCET and the BCET of each process. These times 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 confirmed by RTL simulation. For TL and RTL simulation, the rest of the system is considered as test bench. 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 III 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. 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 that fail. In this case, the early deadline First algorithm generated only 3% of the total number of failings produced by Rate Monotonic.

Thus, 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, because of other additional features, like memory management.

The final delay in the real implementation was higher compared with TL specification. This difference is because of 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.

7. 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.

ACKNOWLEDGMENTS

The authors gratefully acknowledge the support from CNPq/CAPES agencies for R&D and UNISC in the form of scholarships and grants.

REFERENCES

ADOMAT, J., FURUNÄS, J., LINDH, L., AND STÄRNER, J. 1996. Real-time kernel in hardware rtu: A step towards deterministic and high performance real-time systems. In *8th Euromicro Workshop on Real-Time Systems*. L'Aquila. 164–168.

CAI, L. AND GAJSKI, D. 2003. Transaction level modeling: An overview. In *CODES+ISSS*. New Port Beach. 19–24.

CORTADELLA, J. 2000. Task generation and compile time scheduling for mixed data-control embedded software. In *Design Automation Conference*.

DESMET, D., VERKEST, D., AND DEMAN, H. 2000. Operating system based software generation for system-on-chip. In *Design Automation Conference*.

DZIRI, M., SAMET, F., WAGNER, F., CESARIO, W., AND JERRAYA, A. 2000. Combining architecture exploration and a path to implementation to build a complete soc design flow from system specification to rtl. In *ASP-DAC*. Kitakyushu. 219–224.

GAUTHIER, L., YOO, S., AND JERRAYA, A. 2001. Automatic generation and targeting of application-specific operating systems and embedded system software. *IEEE Transaction on CAD*.

GERSTLAUER, A., YU, H., AND GAJSKI, D. 2003. Rtos modeling for system level design. In *DATE*.

GONZALES, M. AND MADSEN, J. 2000. Abstract rtos modeling for multiprocessor system-on-chip. In *International Symposium on SoC*.

GONZALES, M. AND MADSEN, J. 2001. Abstract rtos modeling in systemc. Tech. rep., Denmark.

GROTKER, T., LIAO, S., MARTIN, G., AND SWAN, S. 2002. *System Design with SystemC*. Kluwer Academic Publ., Boston, MA.

KOHOUT, P., GANESH, B., AND JACOB, B. 2003. Hardware support for real-time operating systems. In *CODES+ISSS*. New Port Beach. 45–51.

SILBERSCHATZ, A. AND GALVIN, P. 2000. *Operating System Concepts*. Wiley, New York.

TOMIYAMA, H., CAO, Y., AND MURAKAMI, K. 2001. Modeling fixed-priority preemptive multi-task systems in specc. In *SASIMI*.

WANG, S. AND MALIK, S. 2003. Synthesizing operating system based device drivers in embedded systems. In *CODES+ISSS*. New Port Beach. 37–44.

WOLF, W. 2000. *Computer as Components: Principles of Embedded Computing Systems Design*. Morgan Kaufmann.

YI, Y., KIM, D., AND HA, S. 2003. Virtual synchronization technique with os modeling for fast and time-accurate cosimulation. In *CODES+ISSS*. New Port Beach. 1–6.

Received January 2005; accepted June 2005