Comput Syst Sci & Eng (2011) 6: 425–434

International Journal of
Computer Systems
Science & Engineering

Early, time-approximate modeling of multi-OS Linux platforms in a systemC co-simulation environment

H. Posadas[1], E. Villar[1], Dominique Ragot[2] and Marcos Martinez[3]

[1] *Microelectronic Engineering Group TEISA, University of Cantabria Santander, Spain. E-mail: posadash@teisa.unican.es, villar@teisa.unican.es*
[2] *Thales Communications Colombes, Paris, France. E-mail: Dominique.ragot@fr.thalesgroup.com*
[3] *Design of Systems on Silicon DS2 Paterna, Valencia, Spain. E-mail: Marcos.martinez@ds2.es*

The increase of computational power in embedded systems has allowed integrating together hard real-time tasks and rich applications. Complex SW infrastructures containing both RTOS and GPOS are required to handle this complexity. To optimally map system functionality to the hard-RT SW domain, to the general purpose SW domain or to HW peripherals, early performance evaluations at the first steps of the design process are required. Approximate timed co-simulation has been proposed as a fast solution for system modeling at early design steps. This co-simulation technique allows simulating systems at speed close to functional execution, while considering timing effects. As a consequence, system performance estimations can be obtained early, allowing efficient design space exploration and system refinement. To achieve fast simulation speed, the SW code is pre-annotated with time information. The annotated code is then natively executed, performing what is called native-based co-simulation. Previous native-based simulation environments are not prepared to model multi OS systems, so the performance evaluation of the different SW domains is not possible. This paper proposes a new embedded system modeling solution considering dual RTOS/GPOS systems. A real Linux-based infrastructure has been modeled an integrated into a state-of-the-art co-simulation environment. The resulting solution is capable of modeling and evaluating all HW and SW system components providing the designer with valuable information for early system optimization and design space exploration.

Keywords: Co-simulation, TLM, Approximately-timed, Real-time linux

1. INTRODUCTION

Increasingly embedded system complexity has allowed integrating hard real-time tasks and rich non-RT application together. The combination of these heterogeneous concurrent components interacting among themselves makes the system more difficult to predict and control. As a consequence, more complex infrastructures are required and this is specially important when considering operating systems. Complex embedded SW usually requires considering reuse and integration of third party components, and thus, sophisticated operating systems (OS) are required [1]. Among complex OSes, Linux-based OSes are some of the most commonly used in embedded systems. Linux is a free, open-source OS providing a POSIX-based API. Linux offers powerful and sophisticated system management facilities, a rich cadre of device support, reputation for reliability and robustness, and extensive documentation.

At the same time, electronic designs have to deal with time constrains. Response times, or input and output rates make systems to include real-time characteristics [2,3]. As a consequence, both design tools and platform infrastructures have to be prepared for handling real-time designs. However, a standard general purpose operating system (GPOS) as a Linux kernel cannot support hard real-time tasks. Hard real-time tasks require fa-

cilities to guarantee deadlines are always, in the same way as real-time operating systems (RTOS) do. The use of these RT facilities result in a complete modification in the order tasks are executed and in general in the overall system execution.

In order to combine all required system capabilities together with high efficiency, some electronic systems integrate GPOS and RTOS in the same processor. This solution allows reducing the number of processors required in the system, which minimizes area, power consumption and price. However, the optimization of such complex and flexible platforms requires early system evaluations in order to guarantee that the resulting system has enough computational power to support all the required functionality accomplishing the specified times.

In traditional HW/SW co-design flows, the software development team had to wait to the first hardware prototypes in order to verify and validate the code. As a consequence, evaluation of the whole design was done late in the design process, requiring costly re-design processes when certain catastrophic design errors were detected (i.e. CPU utilization required to be higher than 100%).

To overcome this problem, analytical and simulation techniques have been proposed. Analytical techniques are based on static analysis of software code, considering all possible paths in the CFG (Control Flow Graph). They are usually employed to calculate the WCET (Worst Case Execution Time) for real-time systems. However, estimations obtained using WCET [28] are usually too pessimistic and have to be complemented by simulation techniques.

Simulation techniques are widely used for both functional verification and performance estimation. For such purpose, a large variety of hardware component models and software modeling techniques have been developed. The abstraction level of these models is usually a trade-off between required speed and accuracy.

One of the most employed simulation techniques is the use of an Instruction Set Simulator (ISS) [29]. An ISS reads the binary code compiled for target platform and executes the instructions using a target processor model. However, simulation times are too long for efficient early estimations or design space exploration. Furthermore, ISS systems require the final SW infrastructure, so large engineering effort in porting the OSes is also required.

A first solution proposed to reduce these drawbacks it to use interpreters or as a binary code translators [30]. These techniques achieve shorter simulation times than modeling the full processor internally, but at the cost of providing less accurate results. This is caused by the reduction of internal details considered in the processor models. Nevertheless, this solution is still slow for efficient early estimations, specially when big design spaces have to be covered. Additionally, these solutions also require high porting effort.

To speed up simulation times, approximate time co-simulation techniques based on native execution have been employed. HW description and the $C/C++$ codes of the embedded SW are simulated together using the facilities provided by the SystemC language, a $C++$ library for system modeling. Using this solution embedded software can be directly executed over the host machine, without requiring ISSs or any tool capable of executing target binary code in the host. To obtain performance estimations of SW components, application SW code must be instrumented before the execution, adding information of the performance it is expected the code will have in the target platform. The information required to make the instrumentation can be obtained at source, intermediate or assembly level. The SW infrastructure is based on high-level models of the involved OSs, so porting effort is minimal if the OS models are available. The resulting native execution achieves very fast simulation times, without requiring detailed models of platform components.

In that context, abstract OS models have been proposed for fast time-approximate co-simulation [18, 19, 23, 24]. These OS models provide basic scheduling and communication capabilities for System and HW development oriented environments. However, the effects of hard RT facilities, which have a great impact in SW execution, are not considered in the resulting timeapproximate co-simulations. As a consequence, functional executions and performance estimations obtained without them are potentially wrong.

To solve the previously described problems, the paper presents a complete RT/GP OS model integrated in a SystemC/TLM co-simulation environment. The developed model covers the most important features proposed as RT extensions for Linux, improving performance estimation, system modeling and co-simulation at time-approximate level.

The paper is structured as follows. The next section presents the state of the art in two ways: OS modeling in high-level frameworks, and hard RT improvements in Linux-based systems. Section 3 describes a list of hard-RT improvements required for embedded systems. Section 4 proposes solutions to integrate these features in a SystemC model. Finally an example, results and conclusions are presented.

2. RELATED WORK

2.1 REAL-TIME AND MULTI OSES

Several research works have enhanced the real-time performance of Linux. Firstly, Ingo Molnar developed the real-time preemption patch [4]. This patch adds three main technologies to enhance the real-time performance of Linux, which are IRQ threads, RT mutexes, and high resolution timers [5]. The IRQ thread is a kernel thread handling top-halves of interrupts, which is woken up by ISRs when interrupts occur. In [6], interrupts are also handled by interrupt service tasks whose role is the same as the IRQ threads.

ktimers [9] and UTIME [10] provides optimized implementations for timer resolution in the Linux Kernel. Subsequently George Anzinger introduced the High-Resolution Timers (HRT) patch [15]. Robust mutex implementations has been proposed in FUSYN [11] and Futex [12].

All these proposals have inspired the creation of dual Oss in order to handle properly both real-time and general-purpose applications. For example, Adeos [7] provides a flexible environment for sharing hardware resources among multiple operating systems, or among multiple instances of a single OS. To this end, Adeos enables multiple kernel components, called domains, to exist simultaneously on the same hardware.

The most well-known approach for adding hard real-time capabilities to Linux consists in embedding a dedicated scheduler

aimed at managing time-critical tasks inside the kernel. Several examples can be found.

TimeSys [14], RedHawk [16], RTLinux and its evolution, Enterprise Real-Time Management System (RTMS) [17] provide Linux-based operating systems containing both general purpose and real-time domains. RTAI [8] started from the same approach as RTLinux, but uses a different interrupt virtualization technique, based on the Adeos layer.

In this context, the HYADES [13] system is built over the Adeos layer in order to prioritize hardware interrupt processing, and implement the means of cooperation between the RTOS controller and the Linux kernel. The core of the HYADES realtime system is implemented in an Adeos domain called DIC (i.e. Deterministic Interrupt Computing), embodied in a regular module inside the Linux kernel. Since it is based on RTAI/fusion's core implementation, the DIC controller implements the primary and secondary operation.

2.2 HIGH-LEVEL CO-SIMULATION

Adequate performance estimations are critical when designing a large system. Several solutions have been proposed, including WCET solutions [28, 31], ISS-based simulations [29] and virtualalization [30], each one with different qualities for different design steps and purposes.

Obtaining fast, realistic SW system-level co-simulation has been an important development area in recent years [18-25]. These co-simulations are built on top of system-level languages (SLL) as SystemC [26]. In these high-level simulations, the HW platform is composed of approximate-time SystemC models of the HW components. The SW is simulated though native execution of pre-annotated SW code [27] (Figure 1).

These works usually apply sufficiently accurate time estimations together with OS models. However, OS models oriented to HW-SW co-simulation environments, are usually abstract or partial models. Most of these models are focused on scheduling and provide a minimal set of facilities [18, 19, 23, 24]. These abstract OS models allow the tasks' execution order to be taken into account within the system simulation, providing much more accurate results than only using the standard SystemC facilities.

These OS models do not support a real, complete API. Thus, the application code cannot be refined completely. The resulting code contains abstract system calls, which are not implemented in the real OS, so additional refinement is required to run the SW in the target platform. To avoid these problems, new models based on real RTOS have been proposed [20, 22]. With these OS models, SW refining becomes more efficient. The use of real APIs makes the application code created directly executable on the target platform, reducing the design effort.

To demonstrate the maturity of the area, in [25] a comparison of some of the models presented previously is performed.

Those models present a final limitation. Although application code refinement is mostly supported, Hardware dependent Software (HdS), such as drivers, is not. A more complete model, capable of managing interruption handlers and drivers, is required. Attempts at HW/SW interface modeling have been made at the high level [21, 22]. However, none of the previous models contains hard RT extensions.

Summarizing, there is a lack of high-level simulators capable of modeling all the HW platform components in detail together with OS models containing hard real-time extensions. In this work a solution to overcome this limitation is presented.

To implement the RT modeling infrastructure in SystemC, the solution proposed in this paper is to develop a dual GP/RT OS model, considering the features from the Hyades project. To do so, the Linux-based co-simulation environment proposed in [22] has been extended.

Wherever Times is specified, Times Roman or Times New Roman may be used. If neither is available on your word processor, please use the font closest in appearance to Times. Avoid using bit-mapped fonts if possible. True-Type 1 or Open Type fonts are preferred. Please embed symbol fonts, as well, for math, etc.

3. REQUIREMENTS FOR REAL-TIME MODELING

Soft Real-time tasks are supported in Linux applying different priorities and scheduling modes. However, this solution is not valid for hard RT tasks. For example, deterministic Intensive Computing (DIC) tasks require bounded latencies, reliable execution determinism, and a strict priority management. Their execution quantum must not be significantly perturbed by non real-time activities, which cannot be ensured in that way. Time-critical data acquisition tasks require a complete set of hard real-time features. Guaranteed low interrupt and dispatch latencies are required for these high-priority tasks. Thus, additional hard real-time support is required.

A platform capable of modeling real-time systems, must provide solutions for modeling the performance of the application SW, the effect of the operating systems and the HW platform. Models of the HW platform at multiple levels of abstraction can be found in the literature, so this work is centered on SW code modeling and OS modeling. SW code modeling requires considering the execution times of the cross-compiled code in the target platform and the delays produced by cache misses. At the same time, OS models must include general-purpose and real-time additional support. The RTOS modeling infrastructure developed in this paper extends a previous Linux OS model with a new RT support, allowing the coexistence of both OSes. Thus, it is possible to control the interrupt management in a RT way and then to minimize the latencies (Figure 1).

To provide adequate hard-RT additional support, the following points must be considered:

- A hard RT subsystem must coexist on the same hardware together with the general purpose OS kernel and applications. The coexistence must also allow easy migration of existing real-time applications over the new hard-RT kernel.
- Low-priority interrupt handlers can originally preempt high-priority time-critical tasks, introducing unbounded latencies. Thus, a new interrupt control must be created to intercept, mask and prioritize these interrupts properly.
- Some non RT existing OS services need to be re-implemented to ensure bounded latencies and minimal jitter.

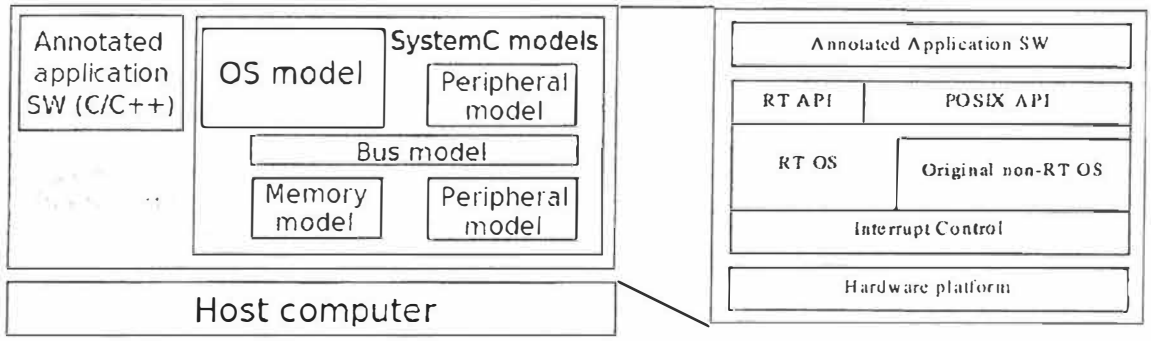

Figure 1 SystemC-based high-level co-simulation and architecture of the OS model proposed.

- System timer precision must be upgraded for time-critical tasks.
- Additionally, new system calls are required to access the new services.

An example of such a service is the standard nanosleep() feature. Its timing precision depends on the period of the system tick. Since the system tick period is usually of the order of milliseconds, exact real-time sleeps cannot be ensured.

As the original OS models a soft, but not a hard real-time system, it will be referred to as "non-RT" infrastructure in the following to simplify the text. The new RT extension will be called the "RT" infrastructure.

4. CODE CHARACTERIZATION

In order to model the performance of the application SW, execution times and cache operation details are added to the original SW code, to transform the functional host execution in an accurate native simulation model. To do so, instrumentation has been used. Instrumentation is a well-known technique which is usually employed to provide extra functionality to a certain application code. This annotated software is communicated in runtime with the cache model and the simulation time manager, so SW execution times and hit/miss rates are estimated.

To accomplish this task, it is necessary to perform a previous characterization of basic blocks in terms of timing and cache behavior. Basic blocks are identified and the number of instructions and cache lines per block are calculated and annotated. Different works at assembly level, intermediate level or source level have been proposed over the last years for obtaining that information. Among them, assembly level provides the most reliable characterization and thus, it has been used in this work. In fact, a hybrid technique is proposed: while basic block identification is performed at source level, characterization is obtained from assembly code. This strategy simplifies the characterization process and speeds up the analysis time. Figure 2 shows an overview of the cache estimation process, including basic block characterization.

Due to the rich syntax of source codes, a $C/C++$ code parser has been developed, so the different elements of the language are easily identified: declarations, statements, expressions, etc. The parser is based in a C/C++ grammar for Bison. The key concept in basic block identification at source level is inserting specific

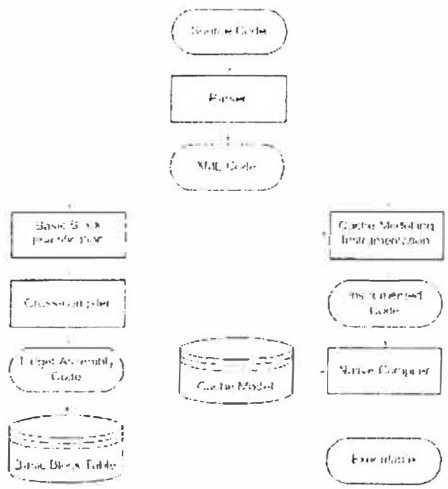

Figure 2 Complete estimation process.

```
...
P1 for (init: cond; step)
{ P2
    body:
P3 } P4
...
```

```
mark_P1:
    init
.L2:
    cond
    bcond .L3
mark_P2:
    body
mark_P3:
    step
    b     .L2
.L3:
mark_P4:
```

Figure 3 Marked code and equivalent assemble.

marks at the beginning and the end of each basic block. This marked code is then cross-compiled, so the marks introduced are preserved in the target assembly code. This procedure guarantees that there is always a direct correlation between source and assembly blocks. Thus, the main questions are: what type of marks should be inserted, and where should they be inserted within source code? The adopted solution is to take advantage of $C/C++$ facilities to mix assembly instructions within source code with the asm sentence. Asm volatile sentences are preserved after compilation, so they are easily identified in the target assembly code. Additionally, to keep the behavior of the original code, the asm instructions inserted consist simply of

labels. Thus, inserted marks looks as:

```
'asm volatile(''mark_xx:'')'
```

A second decision to be taken is where the marks must be placed. As stated before, marks should delimit each basic block at source level. Thus, each $C/C++$ statement requires a custom analysis. As an example, the 'for' statement needs four marks, which are inserted at the key points P1, P2, P3 and P4, represented in Figure 3. Marked code is then cross-compiled for the target processor. The cross-compilation process considers all possible optimizations. The resultant optimized assembly code with equivalent labels is also shown in Figure 3.

The number of instructions of each basic block is easily obtained from the assembly code. L2 and L3 are system labels inserted by the compiler to iterate and exit the loop, respectively. Although compiler optimizations may alter the Control Flow Graph (CFG), labels are preserved in the same order since they have been declared as volatile. The output of this process is a table with block/instruction pairs. This table is used later to characterize each basic block in terms of times and cache lines.

Instructions and data cache modeling requires also static instrumentation, annotating the cache lines required on each basic block, and the corresponding accesses to the cache model for checking if the lines are already in cache or accesses to the main memory are required. More information about cache modeling can be found in [32] and [33].

Compiler optimizations may affect both intra-block and inter-block behavior. Intra-block optimizations are considered in the characterization of the blocks from assembly code. This assembly code already includes both front-end and back-end optimizations. Inter-block optimizations are considered by delimiting the basic blocks at source level. Nevertheless, there are some compiler optimizations which cannot be accurately considered with this technique. Loop unrolling replicates the body of a loop statement in the assembly code, but from source point of view it is a unique block.

Nevertheless, we think that this is a very fast, easy and portable way of obtaining sufficiently accurate estimations for the first steps of the design process, when the platform, the HW/SW partition, resource allocation, etc. are being explored and decided. At the beginning of the design process, the HW and SW codes are usually not the completely optimized final ones. Thus, if the code use for the modeling is not the final one, it can be considered that the effect or these optimizations will result in an error similar or larger than the error provoked by the use of volatile marks. Summarizing, for early modeling, speed and flexibility are much more important at this level than 100% of accuracy.

5. GPOS MODELING

The modeling of a general-purpose operating system requires modeling parallelism, concurrency and other services for communication, synchronization and time management. For implementing them, the POSIX standard has been followed.

5.1 MODELING PARALLELISM OF SW TASKS

Parallelism is modeled by using the SC_THREAD process of SystemC. Therefore, both POSIX processes and threads are modeled in the same way. Thus, the library implements the required actions that give each element its own characteristics. The characteristics of processes and threads are loaded in a list when they are created and these parameters can be modified during simulation using the methods the POSIX standard defines. However, modeling the capabilities derived by the use of separate memory spaces in SystemC is not straightforward.

In order to efficiently support dynamic thread creation, a thread-pool is initialized when the simulation starts. This pool has a predefined number of SC_THREADS (the number can be modified in the source code) which are maintained in a blocked state. During simulation, when a new thread is declared, a thread from the pool is resumed, and stoped again when its functionality is over. Then, the threads can be reused.

5.2 MODELING THE SCHEDULER

Although SystemC provides concurrency support, scheduling is not considered. The SystemC underlying kernel activates in each cycle all the threads that are not blocked, without any consideration about priorities or policies. Thus, a scheduler has been placed on top of the SystemC kernel to ensure that only one thread is executed in each processor at a time. This scheduler ensures that all threads remain blocked, except the one with the greatest priority, which is unblocked. In fact, one thread is unblocked per processor described in the system. The thread executes then until a service from the operating system, such as a semaphore or a mutex, makes it to be blocked again. At this time the scheduler unblocks the next task to execute.

Each execution has two parts. The first one is the functional execution, and the second one is the temporal execution. That is, the code is executed in zero time (in the simulation) and then the thread is slept to take up the corresponding time in the processor, the time annotated in the source code. This time is applied just when a system call is performed. As a consequence, this placement in time is produced before inter-processor communication and synchronizations are made. If during the time the thread is slept, another process with higher priority is awoken, it is executed, to the other process is informed about that preemption. Thus it has to wait to be scheduled again before entering the system call. As a result, when a communication is made, the state of both processes is correct.

However this approach does not model preemption correctly as a SC_THREAD is executed until a wait statement is reached. In order to model preemption adequately, the "wait" function used to sleep the thread and model the execution time automatically returns when another process is awaken. Then, the remaining time is saved and the process waits for the scheduler. When it is resumed, the remaining time is waited and then the process can continue. In Figure 4, an example is used to show the result when using the proposed solution.

In this example, task 1 executes the SW code until the next system calls. At the end, the time accumulated due to execution

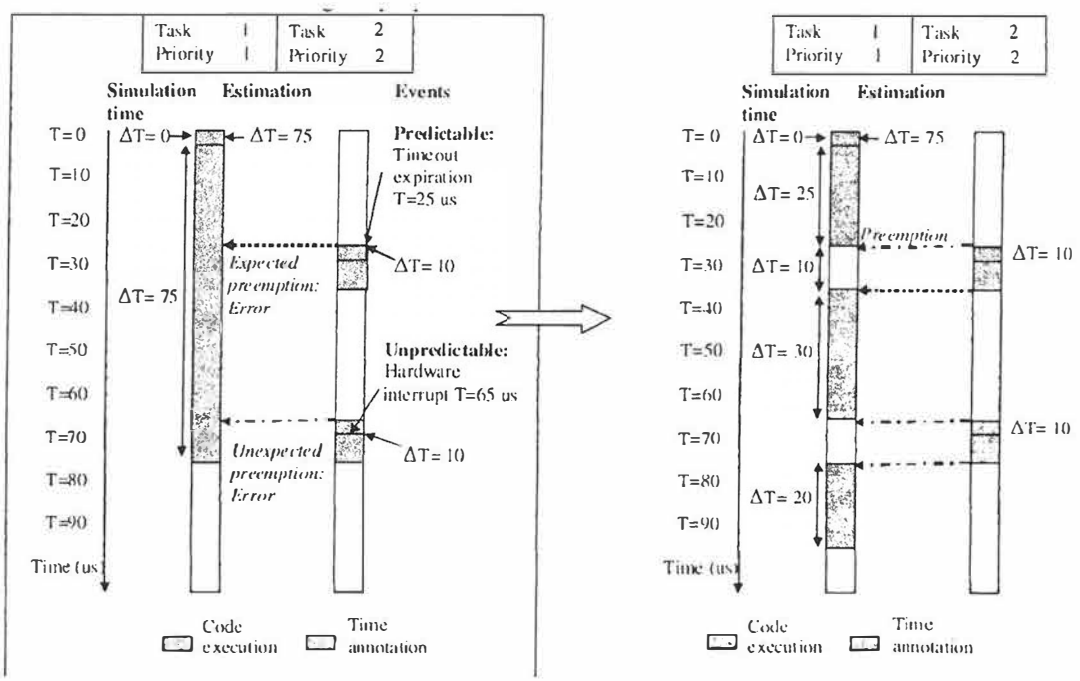

Figure 4 Preemption modeling.

of several basic blocks is 75us. Then, a "wait" function is called for that time. However, at T=25 us, task 2 is awaken, and task 1 has to be preempted. To model that, task 1 is resumed, it calculates that 50us remains to be waited and moves to a blocking state. Task 2 executes, and when it finishes, task 1 is scheduled again, and it waits for the 50 us. But, again it is preempted, remaining 20 us. Thus, the process is repeated again, until all the time is expected, and then the system call can be performed.

This modeling solution does not modify the SystemC kernel. It is based on the use of "wait()" and "notify" SystemC primitives.

5.3 POSIX Interface modeling

POSIX services are provided by the GPOS model in three different ways. Some of them use the underlying host functions, others are completely new, and those that depend strongly on the hardware platform have to be adapted to model correctly its platform-dependent functionality.

If the OS of the host computer is POSIX based, such as UNIX or Linux platforms, some of the host POSIX functions can be reused. These functions are basically those that are platform independent. Mathematical functions, string management, etc., maintain their functionality in every platform and they do not interfere with the scheduler or the parallelism capabilities of the system. Thus, they can be used to model, at least, the platform functionality. To include the timing cost, these functions are wrapped into new functions that take into account the time the function will take in the final processor.

The second group of the API functions is composed of those facilities that allow the designer to interact with the elements that have been implemented in the software execution support described. Parallelism, scheduling, communication, synchronization and timing features are completely platform dependent, so new implementations on top of the SystemC services has been developed.

The last group of POSIX API functions is composed of those functions whose implementation is strongly dependent on the hardware platform. Thus, a general platform execution support model is not possible. Some examples are the I/O functions, which strongly depend on the system drivers, so the implementation cannot be reusable on different platforms. Instead of that, models that allow the designer to simulate the functionality are provided.

Additionally, as required by the POSIX standard, clocks for each process and thread, and for the whole simulation have been implemented. Timers, sleep facilities and alarms are defined by using these clocks. The values of the clocks are updated and the execution time estimated for each code segment. The actions of the elements declared over them, are executed by adding the time each event will take to the events list of the scheduler.

The elements that depend on the real-time clock of the system have been implemented in a different way. With this purpose, a SC_THREAD has been defined that is slept until the next event of that clock is required.

Finally signals have been modeled as defined by the POSIX standard. The signal manager can access the scheduler to allow all blocking communications to implement signals that mean that a thread can be stopped or unblocked independently of the cause that produced this blockage. An additional SystemC thread is used by the signal manager to execute the actions related to delivered signals, since no other process can execute them.

6. HARD RT OS EXTENSION MODELING IN SYSTEMC

Once modeled the POSIX standard in order to provide an GPOS model, the extension developed to support RT characteristics based on the Hyades implementation can be presented. In the real implementation, the two operating systems are placed to run in the same computer over an Adeos infrastructure. Both OSes are mainly independent, with memory separation and different task and resource control. Thus, if a task needs to migrate from one OS domain to the other, it is necessary to have two task control infrastructures, one on each OS, synchronized in some way.

To create an efficient OS model, a different approach is used. Both OS domains are executed within the same memory area in the same host executable program. As there is no physical separation between the OS domains most of the resources and information can be shared between the two domains.

To create the new OS model, the original OS model has been maintained, adding a new hard RT OS infrastructure. To allow easy use of both environments, modifications are hidden to the user as much as possible. The user can program the SW code in the same way, independently of which OS domain the task is in each time. Only the functions for moving tasks from one domain to the other must be explicit in the code.

To change the domain the application must make a system call. The function name in the model has maintained the corresponding Hyades name: pthread_migrate_rt (domain).

Regarding the original OS model, three internal modifications have been applied:

- Scheduling infrastructures have been interleaved creating a single scheduler. When a processor is released or a task is awoken, tasks of the hard-RT domain are selected first, and tasks of the other domain later. This ensures correct selection order.
- Interrupts are shielded and launched by the scheduling system. To do so, interrupts have associated priorities. When a task of the RT domain with higher priority is running, interrupts are delayed.
- Original system calls haves been wrapped. The OS model that must provide the service is selected in the wrapper.

The new hard RT extension covers most of the hard RT improvements presented in section II. In fact all the extensions considered in the Hyades OS have been implemented.

The required mechanisms used to allow the execution of both OS domains can be summarized in three areas: the modeling of the two OS domains and their interactions, the interrupt modeling, and the additional features for RT support, especially latency and jitter reduction.

6.1 OS DOMAIN MODELING

To allow hard RT tasks to be included while maintaining the previous operation mode intact, two OS domains have been modeled, following the real Hyades implementation. However, the overhead caused in the real system by running two completely different OS domains has been minimized in the model. Using separate OS models implies that the act of moving tasks from one to the other is very complex and time consuming. All the task information must be duplicated and the copies must be stopped and resumed depending on which domain the task is currently in.

Thus, the proposed solution is to create a new infrastructure for task scheduling and interruption control, maintaining the original OS model infrastructure as far as possible. Thus, the extended OS has two separate infrastructures containing the implementation of the system calls. However, the elements for task management, such as task creation and destruction, scheduling and preemption mechanisms, are shared. All tasks are created and destroyed as non RT tasks. RT operation mode is only reached when calling the migration system call at run-time.

The scheduler considers two lists of tasks, one for the tasks in the RT domain and the other for the tasks in the non-RT domain. When a new task is required, the first list is accessed. If there is no task ready, the second list is used. Thus, migration only requires changing the list where the task is and modifying the internal task status value where the current domain is indicated.

To control preemptions, interrupt handlers are not launched automatically. Handlers are modeled as high priority tasks and added to the scheduler. When an interrupt is received, a preemption event is always raised, making the current task call the scheduler. However, when the current task is a RT one and has higher priority than the interrupt, it is selected by the scheduler and it can continue. Thus, preemption does not really occur. Furthermore, in this case the time cost associated with scheduler execution is not added to the simulation time. As a consequence, no traces of preemption are included in the model.

This solution enables both models to be handled together, in a simpler way than in the real implementation.

The execution flow of a task is as follows:

- All tasks start as non-RT tasks, running under the original OS domain.
- When it is necessary to enter a real-time section, the task is moved to the RTOS domain, changing the list where it is.
- Finally, when the RT section finishes, the task must return to the non-RT domain.
- When a task is destroyed in a RT section, it is automatically moved to the non-RT domain and then destroyed.

As a consequence, the scheduler has been extended to cover the new tasks' states. New states for the RT tasks have been added to the original non-RT scheduler states (Figure 5). The original non-RT OS model considers 7 states: Created, Ready, SuperUser, Waiting, User, Blocked & Zombie. The new RTOS domain only considers 3 states: Executing, Blocked and Ready (Figure 3). Thus, a RT task has two states, one for the RT domain and one for the non-RT domain.

When a task is moved from the non-RT to the RT domain, the task is moved to Blocked in the non-RT domain, and moved to Executing in the RT domain.

When moving from the RT to the non-RT domain, the operation is similar: the task status in the RT domain is moved to Blocked, and its status in the non-RT domain to Ready. In that

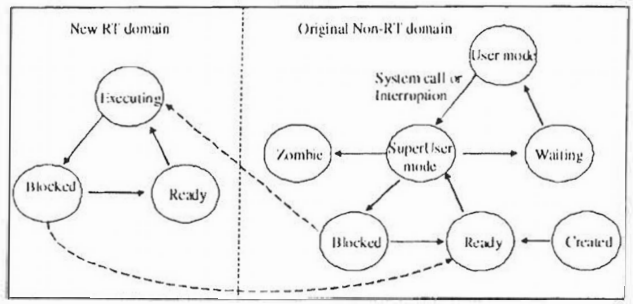

Figure 5 SystemC-based high-level co-simulation.

```
...
if( (next_task = RT_scheduler()) == NULL){ //New code
        next_task = nonRT_scheduler();
        }// New code
    resume ( next_task );
...
```

Figure 6 SystemC-based high-level co-simulation.

way, all tasks are at least blocked in one of the domain lists. This avoids both domains considering the same task to be executable at the same time, thus ensuring correct task scheduling.

To model the execution of both schedulers, the scheduling control in the original OS model has been minimally modified. The call to the scheduling function has been replaced by a two-step process (Figure 6). In the new code the scheduler first searches for a task in the RT list. If a RT task is scheduled, the task is resumed and the non-RT list is not used; otherwise, the non-RT list is called.

6.2 DUAL INTERRUPT SUPPORT

Interrupts are one of the most important risks for hard real-time systems. In a common OS, interrupts are unpredictable. They pre-empt the current task without considering its priority and can provoke priority inversions. Thus, their adequate management is critical for a RT OS. To solve this problem, a double interrupt management level has been integrated in the new OS model. First, the interrupt is processed by the RT interrupt control. If a RT handler has been associated to this interrupt, it is launched considering the handler priority and the priority of the tasks in the RT domain. If a task with higher priority is running, the interrupt is delayed until no tasks with higher priority are ready. If there is no RT handler for the received interruption, the IRQ is delivered to the original non-RT infrastructure, and managed as usual. Since all RT tasks are managed in the RT domain, the original non-RT interrupt management can be maintained.

To implement the double-level interrupt control, the original OS function in charge of the interrupt reception has been modified (Figure 7). First, the presence or not of a RT handler is verified. If not, a non-RT handler is called.

To control system latency and jitter, three main services are provided: high-precision system ticks, system call impersonation and new services.

```
...
if( manage_rt_irq ( irq_number ) == 0){ //New code
        manage_irq ( irq_number );
        }// New code
...
```

Figure 7 SystemC-based high-level co-simulation.

When any timer feature, such as a timer, timeout, alarm or sleep, is used, its accuracy depends on the system tick period. Timer features are managed depending on the tick interrupts. The OS is not capable of considering continuous time advance. It only increases the clocks each time the hardware timer indicates a new period has elapsed. Considering that common ticks have a period of milliseconds, time advances of microseconds or nanoseconds cannot be accurately managed. The system tick period cannot be easily reduced, because the management of all the time features leads to the interrupt handling requiring a significant time. Thus, a new tick strictly for real-time operations is required. As few real-time features are expected to be used simultaneously, the interrupt management latency is very low. Thus, the RT tick interruption can have a very low period without dramatically increasing the system overhead. The tick timer implemented is an a-periodic one. The frequency is set with a user function. Each time the interruption is raised, the interrupt handler must be rearmed.

The second point to be considered is system call impersonation. Some of the POSIX functions managed by the non-RT infrastructure can require special management when used within the RT domain. These functions cover task state changes and accurate time management. Accesses to mutex, fifos, semaphores and other communication channels usually block and unblock tasks. When a task unlocks a mutex, another task blocked in this mutex is unblocked. This produces inconsistencies when using the double domain. When unblocking the mutex, the function in the original OS model moves the task state from Blocked to Ready (in the non-RT domain). However, if the mutex call is done when the task is in RT domain, the operation is incorrect. Instead of moving to the Ready state in the non-RT domain, the state must be moved in the RT domain (Figure 3).

To solve this problem, the function call must detect the domain from which it is called and perform the correct operation. In the proposed model, the solution applied is to modify the OS function in charge of modifying the task state. This function detects the domain where the task is and modifies the state in the corresponding scheduler. This is easier than modifying all the functions directly.

Furthermore, the nanosleep() function has been modified. When called from the RT domain, it uses the RT system tick instead of the common system tick. Thus, the accuracy of the function is automatically increased.

Finally, to manage all this features and to provide some additional RT features, a new API based on the Hyades DIC API has been implemented. Functions have been considered for RT interrupt management, RT task management, RT timing and RT synchronization channels.

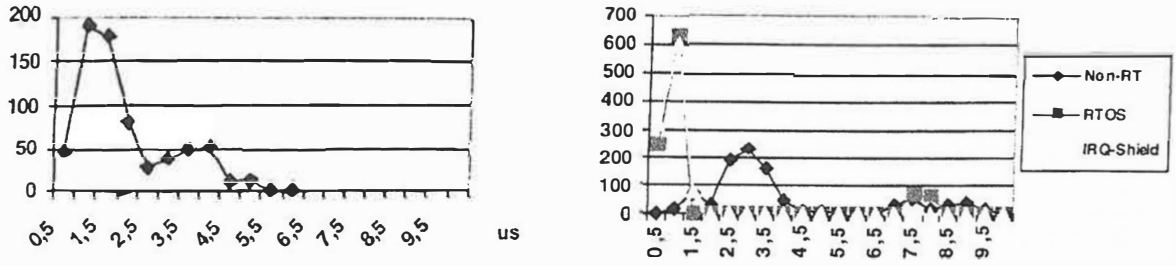

Figure 8 Latency and execution jitter of a SystemC-based high-level co-simulation.

Table 1 Number of cycles measured and estimated.

	Number of cycles					
	Without optimizations (-o0)			With optimizations (-o2)		
	Skyeye	SCoPE	Error (%)	Skyeye	SCoPE	Error (%)
Bubble 1000	30504511	30504511	0	4010006	4510501	12,4812
Bubble 10000	5200180007	5200180007	0	400120008	400130013	0,0025
Vocoder	13466069	14066581	4,45945	6599330	8338713	26,357
Factorial	2747041	2996535	9,08228	1498521	1498518	0,0002
Hanoi	18481575	17695142	4,25523	13107284	11141209	14,9999

Table 2 Simulation times with different configurations.

	Skyeye	Proposed technique		Without data cache		Without caches	
	Time	Time	Speed-up	Time	Speed-up	Time	Speed-up
Bubble 1000	0m2.186s	0m0.028s	×78	0m0.028s	×78	0m0.025s	×80
Bubble 10000	4m6.500s	0m3.486s	×71	0m2.792s	×88	0m1.92s	×130
Factorial	0m1.071s	0m0.014s	×76	0m0.014s	×76	0m0.012s	×90
Hanoi	0m9.426s	0m0.043s	×219	0m0.032s	×294	0m0.020s	×479
Vocoder 10	0m48.793s	0m0.262s	×187	0m0.185s	×263	0m0.105s	×464

7. EXPERIMENTAL RESULTS

To demonstrate the benefits of the RT extension proposed the original OS model and the new extended OS have been compared. This will show how the proposed extensions reduce the system jitter in order to ensure RT capabilities.

The OS models have been tested using the following programs on an ARM926t platform:

- Latency: The latency test measures the latency of the a-periodic timer set at 10kHz frequency.
- Cruncher: Measures the execution jitter of a computation-intensive loop running with or without the RT environment.

The tests were run together with some tasks modeling intensive computing tasks and a network interrupt flood.

In figure 8, we can see the jitter. Applying suitable execution times to the interruptions, OS internal operations and considering the HW platform infrastructure effects, a RT timer with delays less than 5 khz has been obtained. This is much more accurate than the standard timer modeled in the standard POSIX model, which is a 100 Hz timer. Thus, this extension enables the modeling of more time-dependent applications.

Figure 8b also shows that RT extensions outperform Linux by reducing the execution jitter of an intensive data computation. When running the application, the execution time required by the model for this data computation is increased by about 3.5% w.r.t the ideal computation time, with a maximum increment of 10% (Non-RT line).

When applying the RT infrastructure, the mean overhead added by the OS is reduced to a 1%, but some executions are increased by 8% (RTOS line). This overhead is caused by the network interruptions. Thus, when applying the IRQ shield, considering that the data computation has a high RT priority, 8% increments are eliminated, limiting the increment to 2%.

To check the simulation speed and the estimation accuracy of the technique proposed, some small examples and C implementation of 12.2 Kbps GSM Vocoder have been simulated. Results obtained with the proposed technique have been contrasted with ISS simulation (Skyeye) (Tables 1, 2).

As can be shown, the proposed technique achieves high speed up when compared with typical ISS-based co-simulation techniques.

Finally, the reduction of engineering cost for checking the system in different platforms has to be considered. As the OSs used in the technique are SystemC models that run on the host computer, not directly on a target platform, minimal porting is required when exploring different platforms.

8. CONCLUSIONS

A large research effort in real-time extensions for common operating systems, especially Linux, can be found in the literature. As embedded systems are usually RT systems, these extensions may have an important influence on system performance and must be modeled. Current high-level simulation infrastructure considers modeling the OS with sufficiently accurate estimation times. Thus, these models can be extended with new RT features. These RT extensions, although affecting low-level features of the RTOS, are applied at the source-code level. Thus, the resulting technique improves the simulation time. The work shows how native simulation can be accurate enough to model RT features of a RTOS.

Using these extended models, performance estimations of RT systems can be improved in terms of accuracy/speed. As a consequence, the results of design space exploration and system refinement processes, which use the proposed RT modeling infrastructure, could be optimized. The proposed solution for integrating the RT extensions, is to implement a second interrupt control, second RT scheduling and a new set of user space functions. These extensions are placed together with a common OS model in order to obtain a complete system with both real and non-real time capabilities. Native simulation is a powerful technology to efficiently model multi-OS, Linux platforms.

As the implementation is mainly separate from the original OS model and the connection points have been clearly identified, it is possible to apply the solutions proposed in this paper to other simulation infrastructures. As a future work, efficient modeling of platforms executing completely different OSs like Linux and Win32 are being investigated.

ACKNOWLEDMENTS

This work has been supported by the Spanish MICyT and the EC through Complex FP7-249799 and the TEC2008-04107 projects.

REFERENCES

1. R. Lehrbaum, "Using Linux in Embedded and Real-Time Systems", Linux Journal, July 2000.
2. G. Taboada, J. Touriño & R. Doallo, "Performance analysis of message passing libraries on high-speed clusters", IJCSSE, January, 2010.
3. A. Kally et al, " Performance analysis and tuning for clusters with ccNUMA nodes for scientific coputing – a case study", IJCSSE, September, 2009
4. Ingo Molnar real-time preempt patch, http://people.redhat.com/mingo/realtime-preempt/
5. S. Dietrich, D. Walker, "The Evolution of Real-Time Linux", Proc. of Real-Time Linux Workshop, 2005.
6. L. E. Leyva, P. Mejia, and D. de Niz, "Predictable Interrupt Management for Real Time Kernels over conventional PC Hardware", Proc. of the RTAS, 2006.
7. K. Yaghmour, "Adaptative Domain Environment for Operating Systems", http://opersys.com/ftp/pub/Adeos/adeos. pdf
8. L. Dozio, P. Mantegazza. "Real Time Distributed Control Systems Using RTAI", Proc of ISORC, 2003.
9. T. Gleixner. "ktimers subsystem", Linux Kernel Mailing List, 2005. http://lkml.org/lkml/2005/9/19/124.
10. D. Niehaus, R. Menon, S. Balaji, F. Ansari, J. Keimig, and A. Sheth. "Microsecond resolution timers for Linux", 1997. http://www.ittc.ku.edu/utime/.
11. I. Perez, S. Searty, D. P. Howell and B. Hu. "I would hate user space locking if it weren't that sexy...". Proc of OLS, 2004.
12. H. Franke, M. Kirkwood, and R. Russell. "Fuss, futexes and furwocks: Fast userlevel locking in linux". Proc of OLS, 2002.
13. G. Chanteperdrix, A. Berlemont, D. Ragot, and P. Kajfasz, "Integration of Real-Time Services in User-Space Linux", 6th RTL Workshop, 2004.
14. Timesys, http://www.timesys.com/
15. High Resolution Timers project, http://sourceforge.net/projects/high-res-timers
16. RedHawk Real-time Linux, http://www.ccur.com/
17. Enterprise Real-Time Management System (RTMS), http://www.fsmlabs.com
18. Gerstlauer, A. Yu, H. & Gajski, D.D. "RTOS Modeling for System Level Design", Proc. of DATE, 2003.
19. He, Z. Mok, A. & Peng, C. "Timed RTOS modeling for embedded System Design", Proc. of RTAS, IEEE, 2005
20. Hassan M.A., Yoshinori S., K. Takeuchi, Y. & Imai, M. "RTK-Spec TRON: A Simulation Model of an ITRON Based RTOS Kernel in SystemC". Proc of DATE, 2005.
21. Yoo, S. Nicolescu, G. Gauthier L.G. & Jerraya, A.A. "Automatic generation of fast timed simulation models for operating systems in SoC design", Proc. of DATE, 2002.
22. H. Posadas, D. Quijano, J. Castillo, V. Fernández, E. Villar, M. Martínez: "SystemC Platform Modeling for Behavioral Simulation and Performance Estimation of Embedded Systems" in L. Gomes and J. M. Fernandes (Eds.): "Behavioral Modeling for Embedded Systems and Technologies: Applications for Design and Implementation", IGI Global. 2009-07.
23. Hessel, F.; da Rosa, V.M.; Reis, I.M.; Planner, R.; Marcon, C.A.M.; Susin, A.A.: "Abstract RTOS modeling for embedded systems" Proc. of RSP 2004.
24. Schirner G.; Domer, R.: "Introducing Preemptive Scheduling in Abstract RTOS Models using Result Oriented Modeling", Design, Automation and Test in Europe, 2008.
25. Shaout, A.; Mattar, K.; Elkateeb, A.: "An ideal API for RTOS modeling at the system abstraction level", Proc. of ISMA, 2008.
26. SystemC, www.systemc.org
27. J. Schnerr, O. Bringmann, A. Viehl, W. Rosenstiel. "High-Performance Timing Simulation of Embedded Software". In proc. of DAC, 2008.
28. R.Whilhelm, J. Engblom et all. "The worst case execution time problem – overview of methods and survey of tools". ACM Trans. Embedded Computing Systems, 2008
29. ARMulator. http://www.arm.com/support/ARMulator.html
30. QEMU, http://www.qemu.org/
31. R. Obermaisser, C. El-salloum, B. Huber, H. Kopetz, "Modeling and Verification of Distributed Real-Time Systems using Periodic Finite State Machines", IJCSSE, July, 2008
32. J. Castill, H. Posadas & E. Villar, "Fast instruction cache modeling for Approximate Timed HW/SW co-simulation", in proc. of GLSVLSI'10
33. H. Posadas, L. Díaz & E. Villar, "Fast data cache modeling for native co-simulation", in proc. of ASP-DAC'11.