A Real-Time Application Design Methodology for MPSoCs

Giovanni Beltrame
European Space Agency
Noordwijk, The Netherlands
giovanni.beltrame@esa.int

Luca Fossati, Donatella Sciuto
Politecnico di Milano
Milano, Italy
{fossati, sciuto}@elet.polimi.it

Abstract

This paper presents a novel technique for the modeling, simulation, and analysis of real-time applications on Multi-Processor Systems-on-Chip (MPSoCs). This technique is based on an application-transparent emulation of OS primitives, including support for RTOS elements. The proposed methodology enables a quick evaluation of the real-time performance of an application in front of different design choices, including the study of system's behavior as tasks' deadlines become stricter or looser. The approach has been verified on a large set of multi-threaded benchmarks. Results show that our methodology (a) enables accurate real-time and responsiveness analysis of parallel applications running on MPSOCs, (b) allows the designer to devise an optimal interrupt distribution mechanism for the given application, and (c) helps dimensioning the system to meet performance and real-time needs.

1 Introduction

Increasingly large portions of electronic systems are being implemented in software, with the consequence that its development effort is becoming a dominant factor in the development flow. The problem is made worse by the strong trend from the "classical" embedded systems towards *real-time* (RT) systems with explicitly concurrent hardware, more difficult to analyze and to model. In such systems, beyond the correctness of algorithms, early verification of the real-time behavior and timing constraints is essential. Guaranteeing the required properties with explicitly concurrent software and hardware adds a degree of complexity, coupled with the fact that software deployment and testing on target hardware is difficult and time consuming.

To gather timing details and to validate the functionality of the overall system as soon as possible in the development process, high level models of the interactions among application, operating system, and hardware platform are needed. Unfortunately, generally used methodologies suffer from the *code equivalence* problem (as presented in [12]): the code executed by the virtual system is different from the code executed by the final deployed hardware, especially for what concerns OS primitives. This may change the overall system behaviour leading to less-than-optimal or wrong design choices. RT systems further complicate the situation as the correctness of the computation is highly dependent on its timing behavior; this implies accurate modeling of scheduling choices, task interactions, and interrupt response times. Moreover, the implementation of critical RTOS parts can be carried out either in software or using hardware accelerators, with major consequences on system cost, behavior and timing features.

In this work we provide a co-design environment suitable for the development multi-processor systems with real-time requirements. The base idea consists in the *transparent emulation* of RTOS primitives on top of a virtual platform described at transaction level; the implementation guarantees full compatibility with any POSIX-compliant application. Overall, our approach provides fast and accurate simulation results, allowing effective high-level design space exploration for multi-core RT systems. Our methodology can be applied to a variety of tasks, such as analysis of system responsiveness in front of different load and of varying frequency of external events, and as exploration of different scheduling policies.

This paper is organized as follows: Section 2 describes previous research on the subject and Section 3 presents how the proposed methodology addresses the identified issues. Finally, Section 4 shows the experimental results and Section 5 draws some concluding remarks.

2 Previous work

HW/SW co-design flow [11] usually starts at system-level, when the boundaries between the hardware and software parts of the final system have not yet been established. After functional verification, the HW/SW partitioning takes place and co-simulation is employed to validate and refine the system. The tight time-to-market constraints, the high

complexity of current designs, and the low simulation speed of Instruction Set Simulators (ISSs) push for the addition of (RT)OS models in system-level Hardware Description Languages (HDL) (as SystemC or SpecC). This allows native execution of both the hardware and software models of the system, consistently speeding-up simulation. In addition, as both hardware and software partitions are described using the same HDL, it is easy to move functionalities between them. Such ideas are presented in [8, 6, 3, 9]: these works model the application, the hardware, and the services of the (RT)OS using the same HDL. Due to limitations in the typical HDL processing model, true concurrency is not achieved, and a trade-off has to be determined between simulation speed and accuracy of the inter-task interactions models. When the design is refined, the RTOS model can be translated automatically into software services; unfortunately in most real-world situations the use of a widely adopted RTOS is preferred, meaning that results taken during the modeling phase are no longer accurate.

An extension to these works has been implemented by Schirner and Domer [10] addressing the problem of modeling preemption, interrupts, and inter-task interactions in abstract RTOS models, but they mainly concentrate on simulating the system timing behavior and the code equivalence problem is not taken into account.

He et al. [4] presents a configurable RTOS model implemented on top of SystemC: as opposed to other approaches, only the software part of the system is modeled, while the hardware portion is taken into account just through timing annotations inside the RTOS model.

A different technique is employed in [12, 5], for automatic generation of timed OS simulation models. These models partially re-use unmodified OS primitives, thus mitigating the code equivalence problem. High emulation speed is obtained thanks to native execution on the host machine, but the timing of the target architecture is not accurately replicated and it does not allow precise modeling of multi-processor systems. On the contrary, we use ISSs in our approach: this means lower simulation speed but also, as the assembly code of the final application is used, minimization of the code equivalence problem. Our approach is also OS-independent, enabling wider design space exploration.

These approaches are valid to help the designer perform and refine the HW/SW partition, but they do not help in the validation of the high-level design (for the code equivalence problem), and they are limited in the assessment of the system's timing properties. However, the execution on an ISS of the exact same software which will be deployed on the embedded system is seldom possible because the RTOS has to be already chosen and ported to the target hardware, meaning that it might be difficult or impossible to refine the HW/SW partitioning (since the OS should be updated accordingly) and to explore the other system configurations.

In this paper we propose a way to emulate RTOS primitives in order to minimize *code equivalence* issues while still maintaining both *independence from specific OSes* and *high timing accuracy*.

3 Proposed Methodology

For the implementation and the evaluation of the design methodology presented in this paper we use the Open Source simulation platform ReSP [1].

In this work, we exploit and extend ReSP's System Call Emulation subsystem [1] to support the analysis of real-time systems and applications. The presented functionalities are used for preliminary exploration of the applications' behaviour, for guiding the designer in the choice of the target RTOS and as a support for early HW/SW codesign. The rest of this Section is devoted to a more thorough explanation of such concepts.

3.1 System Call Emulation

System Call Emulation is a technique enabling the execution of application programs on an Instruction Set Simulator (ISS) without the need to simulate a complete OS. The low level calls made by the application to the OS routines (*system calls*, SC) are identified and intercepted by the ISS, and then redirected to the host environment which takes care of their actual execution. Suppose, for example, that the simulated application program contains a call to the `open` routine to open file *"filename"*. Such a call is identified by the ISS and routed to the host OS, which actually opens *"filename"* on the host's filesystem. The file handle is then passed back to the simulated environment. A simulation framework with System Call Emulation capabilities allows the application developers to start working as early as possible, even before a definite choice about the target OS is performed. This can also help in the selection, customization, configuration, and validation of the OS itself.

Figure 1 shows ReSP's innovative System Call emulation mechanism where each ISS communicates with two components: the centralized Trap Emulator (TE) and the Concurrency Manager (CM); these are the components responsible for forwarding the System Calls from the simulated to the host environment, and for managing task creation, termination, and synchronization. Instead of identifying the SCs through particular assembly instructions or special addresses (as in most simulator environments), we use the name (i.e. the symbol) of the corresponding routine. When the application program is loaded, the names of the low level SCs (e.g. `sbrk`, `_open`, etc.) are associated with their addresses in the binary file and registered with the TE. At runtime the ISS then checks for those addresses: when one is found, the corresponding SCs is emulated on the host

environment. In addition to System Call emulation, these functionalities can be used, through the CM, for the emulation of concurrency management (thread creation, destruction, mutex lock, unlock, etc.) routines. With respect to previous works, these mechanisms show the following advantages:

1. *Independence from the cross-compiler toolchain*: since the names of the system call routines are used, there is no need to adhere to the conventions with which the software is built or to create fictitious jumps in the code
2. *High interoperability with different ISS types*: the IF is the only component which needs customization to allow a new ISS to be integrated with the TE
3. *Extensibility*: the presented mechanism can also be used for preliminary hardware/software partitioning. Moreover, by emulating the POSIX-Threads routines, multi-threaded applications can be easily simulated.

Since only the low level SCs (e.g. `sbrk`) are emulated and the rest of the OS code (e.g. `malloc`) is executed unmodified in the ISS, our method maintains high code equivalence with the final software, even at the assembly level.

In this paper, the TE and a modified CM (see Section 3.3) are used to model real-time tasks and analyze the target application's real-time performance.

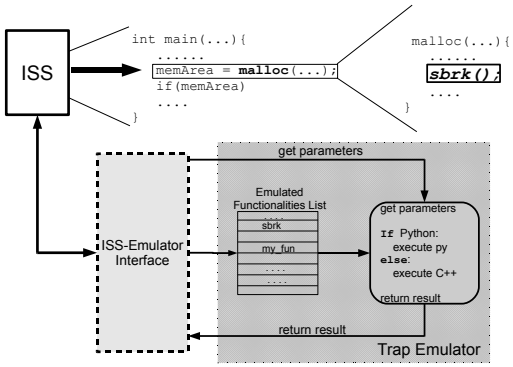


Figure 1. Internal structure and working mechanisms of the function trap emulator

3.2 Pthreads as a Real-Time concurrency model

PThreads are a well known concurrent application programming interface (API) and, as part of the POSIX standard, are available for most operating systems (either natively or as a compatibility layer). The Pthread API provides extensions for managing real-time threads, in the form of two scheduling classes:

fifo : threads of equal priority are scheduled following a first-come first-served policy; if a thread of high priority is created while one of lower priority is running, the running thread is preempted.

round-robin : same scheduling policy of *fifo*, with the difference that the processor is shared, in a round robin fashion, among threads of equal priority.

In order to manage these functionalities, PThreads provide routines for setting/reading/changing thread priorities and scheduling policies. However, even when using POSIX-Threads RT extension, the standard does not fully allow the management of RT systems. Important features, such as task scheduling based on deadlines, are not present and this prevents an effective modeling and analysis of a wide range of RT systems. For this reason our emulation layers extends the POSIX-Thread standard with the introduction of the *Earliest Deadline First* (EDF) [7] scheduling policy and with the possibility of declaring a task as un-preemptible. However, theoretical results [2] expose that EDF scheduling brings better performance with respect to standard priority based scheduling. In our implementation, the emulated RT features are compatible with the popular OS RTEMS task management policies.

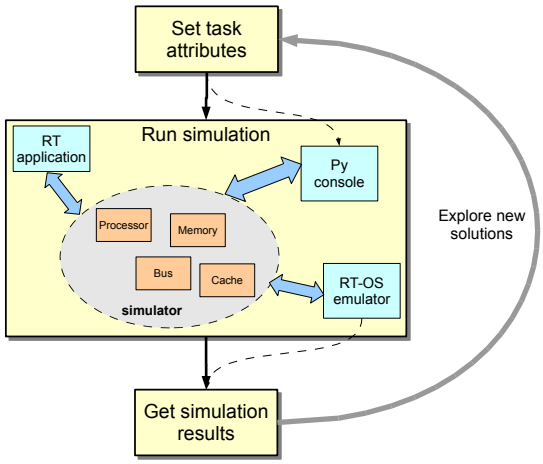


Figure 2. Exploration flow of RT policies: note how task attributes are modified at runtime throught Python's cosole.

The main objective of this work consists in enabling exploration, tuning and analysis of RT systems: in order to effectively and efficiently perform this, we need to be able to explore the task scheduling policies, their priorities, and in general, tasks' attributes. As modifying the source code is not suitable to fast co-exploration, thread attributes and scheduling policies can be specified also *outside* the simulations space, using ReSP's capabilities. This gives the designer two possible alternatives: specifying the desired RT behaviour directly in the application's source code, or via Python scripting. This means that the system load and RT behaviour can be modified during simulation, enabling an effective exploration of the system's real-time behavior.

The former mechanism is simply obtained by emulation of all threading related primitives; in particular the calls made by the application software to the functions for managing thread attributes are redirected to the CM, which takes care of managing and scheduling the tasks according to such attributes. The latter method, consists of using Python to directly export the internal structure of the CM to ReSP. As such, it is possible, either before or during the simulation, to modify the CM status and change the thread management policies without modifications to the application's source code.

3.3 A Real-Time Concurrency Manager

The TE was extended for the emulation of concurrency management routines with an additional unit, called *Concurrency Manager*. The overall mechanism is analogous to the one depicted in Figure 1, but instead of trapping I/O or memory management, the TE traps routines for thread creation, destruction, synchronization etc. During execution, all calls to `pthread` routines are trapped and forwarded to the Concurrency Manager in the simulator environment. If the application software is compiled with a recent GNU GCC compiler (at least version 4.2) it is also possible to successfully emulate OpenMP directives.

This concurrency manager was augmented to deal with Real-Time extensions and to correctly keep statistics about issues such as missed deadlines, serviced interrupts, etc.

Real-Time scheduler: we implemented it in three different versions: *FIFO*, *Round-Robin*, and *Earliest Deadline First* (EDF). Each task can be assigned a scheduling policy and tasks with different policies can co-exist in the system. Tasks' policies can be varied at runtime either from the application code or by directly interacting with ReSP through the Python console. The latter mechanism has been implemented to enable flexible task management, thus allowing an effective and efficient exploration of the different scheduling policies and priorities, and the different RTOS configurations. Tasks with the EDF policy are assigned the highest priority. The scheduler is able to manage shared memory platforms with an arbitrary number of symmetric processors. Since scheduling and, in general, task management operations are performed in the host environment, it is possible to add features such as deadlock and race-conditions detection without altering the system's behavior, as it would, instead, happen if such features were implemented inside the simulated software. For this reason, our system can be also successfully used for verification of the system correctness.

Interrupt Management: it is composed of an emulated interrupt generator and and Interrupt Service Routine (ISR) manager. While the former is present only to emulate external events and to force execution of ISRs (to enable the analysis of the system behavior under different realistic environmental conditions), the latter feature is used to deal with ISRs no matter how they are triggered. No major modifications were necessary to the system to control ISRs since, after creation, they are treated as standard real-time tasks.

Python Integration: this features enables tasks control from outside of the simulated application. This means that from ReSP's interactive shell it is possible to manage task priorities, deadlines, etc. As such there is no need to modify the simulated software to perform an effective exploration and to analyze the effects of different scheduling policies and/or priorities.

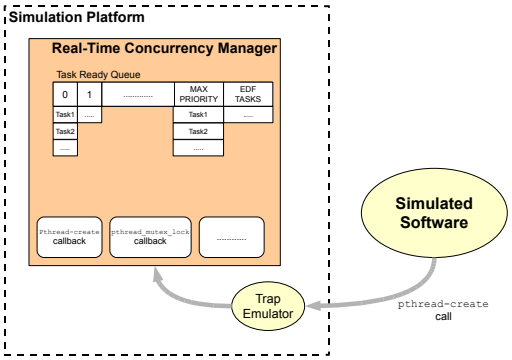


Figure 3. Detailed Structure of the Real-Time Concurrency Manager (CM): through the TE the CM communicates with the simulated application. in the CM RT-tasks are organized in queues of different priorities.

4 Experimental Results

The methodology has been tested on a large set of OpenMP-based benchmarks (namely the OMPScr suite) and a large parallel application, namely ffmpeg (video encoding/decoding).

The basic assumption of this work is that the system is subject to a mixed application workload: a computationally intensive element with soft real-time constraints, and a set of elements with very strict hard real-time characteristics, here called computational and real-time parts, respectively. The number and parameters of both the computational and real-time parts varies and strictly depends on the system being considered. This model well represents applications such as observation spacecraft payload, where massive data processing is needed with high availability, while response to external stimuli within a given time is paramount (for example for the spacecraft's navigation system).

The purpose of our methodology is to answer a set of key questions during the development of real-time applications running on an MPSoC:

1. What is the performance of the real-time applications? Is the system missing any deadlines with the current hardware and scheduling setup?

2. What is the performance of the computational part? Is it performing within requirements?

3. How much performance can the current hardware and software setup deliver? Is it possible to add additional computational or real-time tasks without affecting global performance? Can we reduce the number of hardware resources? What is the benefit of moving parts of the application or OS to hardware?

This leads to two sets of experiments that can be easily performed using the proposed methodology, as shown in the following.

All tests have been executed using ReSP on a multi-ARM architecture consisting of a variable number of cores with caches, and a shared memory, all interconnected by a shared bus. Simulations where timing was recorded were run on a Core 2 Duo 2.66GHz Linux machine.

4.1 Real-Time Performance Evaluation

Using the proposed methodology, a designer can verify the real-time performance of a multi-processor system under load and explore the use of different interrupt distribution and handling schemes. As proof-of-concept, we ran the benchmarks as computationally intensive applications, while the real-time tasks are implemented by synthetic functions, with varying deadlines. These functions can be categorized as: (a) housekeeping: scheduled regularly, perform sanity checks, repetitive tasks, etc., and (b) response to external events: when an alarm is fired, its response is usually required within a given deadline.

A first analysis that is performed with the current methodology is to run the real-time part separately from the computational part, reducing all OS related latencies (such as the latency of the `mutex lock` operation) to zero: the obtained *concurrency profile* shows the number of active PEs in time, i.e. the effective utilization of the system resources. A similar graph is derived for the computational part, allowing the designer to determine if sufficient resources are available to run the application within its performance constraints. Finally, the computational and real-time parts are combined together and the concurrency profile is drawn as shown inf Figure 4. This graph helps the designer tweaking the hardware and software to match the desired requirements. As an example, the combination diagram (Figure 4(b)) can show a lower-than-expected utilization in case

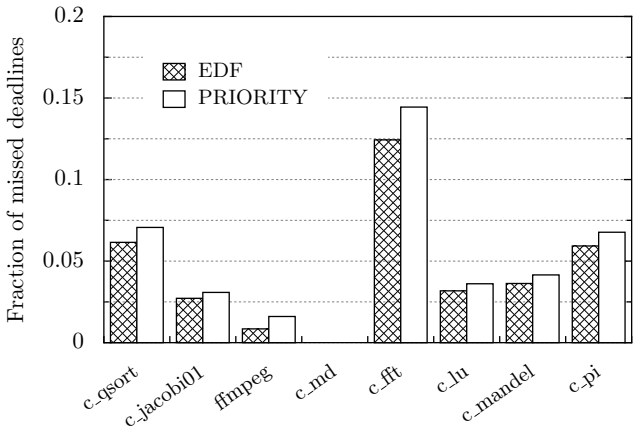


Figure 5. Fraction of missed deadlines with different schedulers and high RT workload (1kHz)

access to a shared bus represents a bottleneck in the system. If, instead, utilization is already at a maximum, the designer can conclude that more processing elements are needed to reach the performance requirements. Simulating the system with realistic OS-related latencies (that can be targeted to any possible OS choice) leads to determining the best OS choice for the current application.

Figure 5 shows how the methodology is used to determine the best scheduler for the system: the performance is graphed for two schedulers (PRIORITY and EDF).

4.2 Computational Performance Evaluation

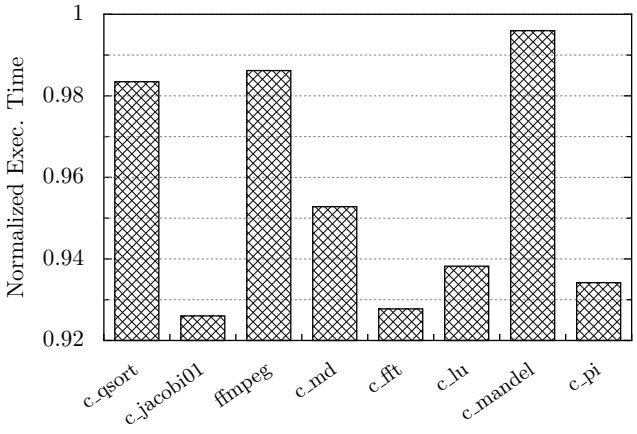
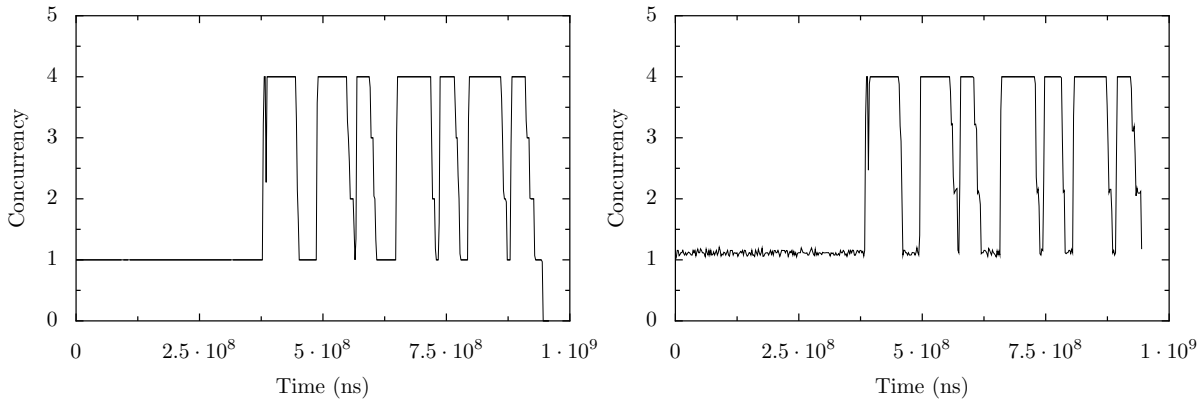


Figure 6. The performance impact of the RT part on the computational part, i.e. their relative execution time when compared to execution without the RT part

Running the application with and without RT tasks



(a) Computational Part

(b) Complete System

Figure 4. The concurrency profiles of the ffmpeg benchmark, showing the computational parts ((a)) and the combination of the computational and real-time parts ((b))

shows the different computational performance of the system, as depicted in Figure 6. As in our methodology the RT or non-RT status of a task can be changed without modifications to the code, this evaluation is simply made with two runs of the simulator. The designer can see how changes in the OS scheduling affect the performance of the system.

5 Concluding Remarks

In this paper we presented an innovative mechanism for Operating System emulation inside Instruction Set Simulators. In addition to being non intrusive in the ISS source code, the described techniques are extended for the emulation of real-time tasks. High code equivalence is maintained, enabling fast and accurate simulation of real-time applications.

Our methodology has been applied to many mixed-workload (real-time and non real-time) applications and benchmarks, showing how it can be used to analyse their real-time behavior and how the system can be sized to obtain the desired peformance for both the real-time and non-real time parts.

References

[1] G. Beltrame, L. Fossati, and D. Sciuto. Concurrency emulation and analysis of parallel applications for multi-processor system-on-chip co-design. In *CODES/ISSS '08*, pages 7–12, New York, NY, USA, 2008. ACM.

[2] G. C. Buttazzo. Rate monotonic vs. edf: judgment day. *Real-Time Syst.*, 29(1):5–26, 2005.

[3] A. Gerstlauer, H. Yu, and D. Gajski. Rtos modeling for system level design. In *DATE, 2003*, pages 130–135, 2003.

[4] Z. He, A. Mok, and C. Peng. Timed rtos modeling for embedded system design. In *RTAS 2005*, pages 448–457, 2005.

[5] S. Honda, T. Wakabayashi, H. Tomiyama, and H. Takada. Rtos-centric hardware/software cosimulator for embedded system design. pages 158–163. IEEE Computer Society, 2004.

[6] E. Huck, B. Miramond, and F. Verdier. A modular systemc rtos model for embedded services exploration. In *DASIP 2007*, 2007.

[7] M. Kargahi and A. Movaghar. Non-preemptive earliest-deadline-first scheduling policy: A performance study. In *Proceedings of the 13th IEEE ISMASCTS*, pages 201–210. IEEE Computer Society, 2005.

[8] R. L. Moigne, O. Pasquier, and J.-P. Calvez. A generic rtos model for real-time systems simulation with systemc. In *DATE*, page 30082. IEEE Computer Society, 2004.

[9] H. Posadas, J. Ádamez, P. Sánchez, E. Villar, and F. Blasco. Posix modeling in systemc. pages 485–490, Yokohama, Japan, 2006. IEEE Press.

[10] G. Schirner and R. Domer. Introducing preemptive scheduling in abstract rtos models using result oriented modeling. 2008.

[11] W. Wolf. A decade of hardware/software codesign. *Computer*, 36(4):38–43, 2003.

[12] S. Yoo, G. Nicolescu, L. Gauthier, and A. Jerraya. Automatic generation including fast timed simulation models of operating systems in multiprocessor soc communication design. 2002.