A Mixed Level Simulation Environment for Stepwise RTOS Software Refinement

Markus Becker, Henning Zabel, and Wolfgang Mueller

University of Paderborn/C-LAB,
{beckerm, henning, wolfgang}@c-lab.de

Abstract. In this article, we present a flexible simulation environment for embedded real-time software refinement by a mixed level cosimulation. For this, we combine the native speed of an abstract real-time operating system (RTOS) model in SystemC with dynamic binary translation for fast Instruction Set Simulation (ISS) by QEMU. In order to support stepwise RTOS software refinement from system level to the target software, each task can be separately migrated between the native execution and the ISS. By adapting the dynamic binary translation approach to an efficient but yet very accurate synchronization scheme the overhead of QEMU user mode execution is only factor two compared to native SystemC. Furthermore, the simulation speed increases almost linearly according to the utilization of the task set abstracted by the native execution. Hereby, the simulation time can be considerably reduced by cosimulating just a subset of tasks on QEMU.

1 Introduction

The introduction of RTOS models raised the level of software-aware abstraction to true electronic system level designs. Today, real-time properties of even very complex designs with several CPUs can be verified efficiently. For this, RTOS models typically wrap the native execution of functionally segmented C code in a system level design language like SystemC[1] or SpecC[8]. Hereby, RTOS services are provided by means of an application programming interface (API) and task synchronization is achieved by implementing dedicated real-time scheduling policies. For timing analysis and estimation, software is partitioned into segments which are back annotated by timing information.

For this, the execution time of software tasks, functions, and basic blocks, i.e., linear code segments followed by a branch instruction, is either measured on the target CPU or retrieved from a static timing analysis. Sometimes, the timing information for the functional segments is not available as due to intellectual property protection the C code is not accessible. Then, an abstract RTOS simulation which requires the partitioning and annotation of the code cannot be applied. In such a case, the application software and RTOS has to be completely simulated by an Instruction Set Simulator (ISS). Since an ISS usually comes with a slow execution speed, it is not possible to efficiently perform detailed analysis.

Therefore, we developed a new approach to combine the benefit of the ISS with the speed of the RTOS abstraction. Our approach applies a clear separation of the software and the operating system along the natural interface of system calls and to clearly distinguish application software from the operating and its services. Application software typically runs in user space with unprivileged user mode access. Only through system calls the software can get kernel mode access to the kernel space. In contrast, system services and drivers have to run in kernel space for kernel mode access for the execution of privileged instructions for full access to the hardware.

To reach a high simulation speed, we use the open source QEMU software emulator [5] for instruction set simulation in combination with SystemC since it provides fast execution of cross-compiled target code due to an advanced dynamic binary translation. Combined with a fast execution time estimation approach and an efficient synchronization scheme, we achieve a much higher simulation speed than cosimulations of SystemC with a traditional ISS.

In the first refinement levels, the kernel space is abstracted by an abstract RTOS model. By combining it with user mode QEMU emulations, for each user space task it can be separately decided whether to be executed natively or to be coexecuted on the target Instruction Set Architecture (ISA) under QEMU. As such, software task refinement from system level towards firmware is smoothly supported as depicted by Figure 1. The task refinement starts from a) native SystemC and is then refined via b) mixed user space (i.e., subsets of tasks coexecuting on ISS) to c) the user space emulation (i.e., all tasks coexecuting on ISS) and finally arriving at d) the full system emulation including the complete target RTOS kernel and kernel space drivers.

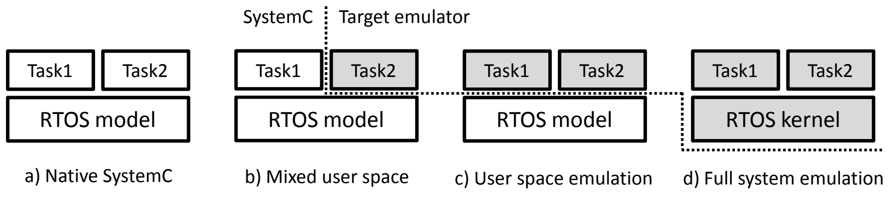


Fig. 1. RTOS software refinement.

The remainder of this article is organized as follows. Section 2 describes recent research in the field of RTOS simulation. Section 3 introduces the concepts of our simulation environment providing basic concepts of the mixed level simulation. Section 4 underlines the feasibility and efficiency of our mixed level simulation levels with some experimental results. Finally, Section 5 concludes with a summary.

2 Related Work

We can find several related work in the areas of RTOS simulation and RTOS software refinement methodologies.

Early work by Hassan et al. [11] outlines a simple RTOS simulation in SystemC, where specific schedulers can be derived from a basic class. They model processes by a 1-safe Petri-Net with atomic transitions annotated by time and power consumption.

Individual state transitions are triggered by the μ-ITRON-OS-based RTOS kernel μ-Itron via round-robin scheduling, and I/O operations call hardware operations via a bus functional model. They do not consider interrupt management.

Krause et al. [12] present a tool-based approach for system refinement with RTOS generation. Stepwise refinement covers abstraction levels from CP (Communicating Processes) to CA (Cycle Accurate) models. In the context of the PORTOS (POrting to RTOS) tool, they introduce the mapping of individual SystemC primitives to RTOS functions. Mapping to different target architectures is implemented by a macro definition. PORTOS is configured by a XML specification characterizing the individual target platforms.

Destro et al. [7] introduce a refinement for multi-processor architectures in SystemC with a mapping from SystemC primitives to POSIX function calls. Starting from functional SystemC, first processor allocation and then HW/SW partitioning are performed. A final step maps SystemC to a cosimulation of hardware in SystemC and software running on top of an RTOS. After the mapping hardware threads are executed by a specific SystemC compliant hardware scheduler.

Posadas et al. [15] have published several articles on RTOS simulation. They introduce concepts of their freely available SystemC RTOS library PERFidiX, which covers approximately 70% of the POSIX standard. They report a gain in simulation speed w.r.t. ISS of more than 142 times in one of their first publications, including a 2x overhead in speed due to their operator overloading.

RTOS simulation with time annotated segments is either based on on a standard RTOS API, like the previous approach, or an abstract canonical RTOS. Gerstlauer et al. [9] implemented a canonical RTOS in SpecC. More details of that SpecC library were outlined by Yu [18], who also introduced an approach for SoC software development and evaluation of different scheduling algorithms and their impact on HW/SW partitioning in early design phases. Communication between tasks, including interrupts, is based on events. ISRs are modeled as tasks. Since task scheduling is implemented on top of the non-preemptive SpecC simulation kernel, simulations may give inaccurate results, which has most recently been resolved by Schirner and Doemer [16].

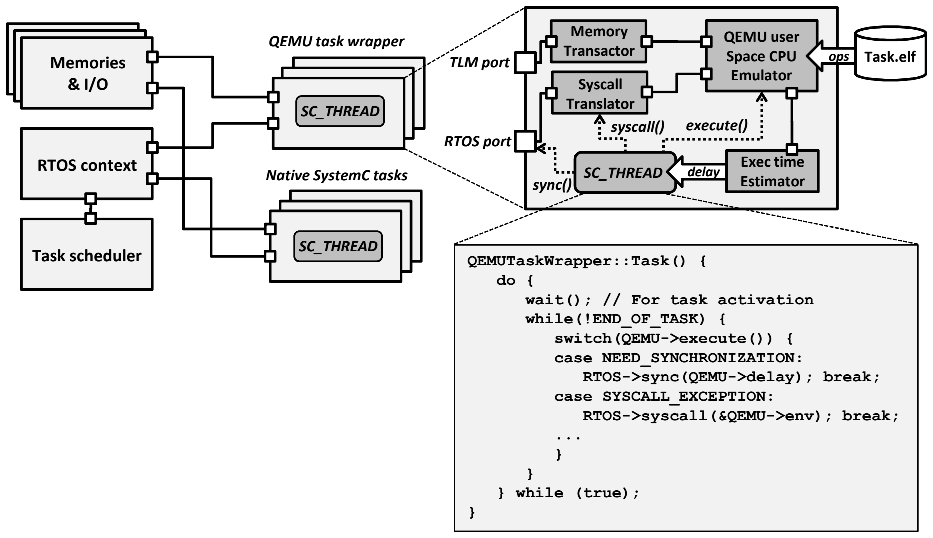
However, interrupts are still modeled as high priority tasks and have to apply the same scheduling algorithm as the software scheduler. Our SystemC RTOS model [19] follows [9] but overcomes the limited interrupt modeling accuracy by means of providing dedicated schedulers for tasks and ISRs.

In [3] we additionally proposed a four level RTOS and communication refinement flow for TLM2.0-based designs comprising our SystemC RTOS model and the QEMU system emulator for ISS cosimulation. There are some other existing approaches using QEMU dynamic binary translation for ISS. For instance, the GreenSoCs project QEMU-SystemC [14] combines SystemC and QEMU in a HW/SW cosimulation by providing a TLM interface for device driver development of memory mapped SystemC HW descriptions. In [10] the authors extend the QEMU dynamic binary translation by an approximate cycle-count estimation for fast performance evaluation in MPSoC designs. They also consider precise simulation of cache effects in MPSoCs by substituting the internal memory model of QEMU with an external cache and memory model in SystemC.

Some work can be found considering the combination of RTOS models and ISS. For instance, Krause et al. [13] combined an abstract SystemC RTOS model with the SimpleScalar ISS for target software evaluation. For this, they provide a virtual prototyping environment that abstracts the scheduling and context switching by their SystemC RTOS model whereas the residual software parts keep running on the cosimulated SimpleScalar ISS.

3 RTOS Simulation Environment

We introduce a mixed level RTOS-aware cosimulation environment with a refinement from an abstract RTOS model towards a cycle-accurate simulation of the instruction set on the basis of a per-task refinement, i.e., an independent refinement of each application task from the abstract description in SystemC towards a target-specific firmware binary. At the same time, the refinement of abstract RTOS services to the target RTOS is provided by a stepwise migration of RTOS primitives like task scheduling, I/O, and task communication from the abstract RTOS model to user mode emulation. Finally, the abstracted kernel is seamlessly replaced by the kernel of the target operating system in order to run a full system cosimulation.



```
QEMUTaskWrapper::Task() {
   do {
      wait(); // For task activation
      while(!END_OF_TASK) {
         switch(QEMU->execute()) {
         case NEED_SYNCHRONIZATION:
            RTOS->sync(QEMU->delay); break;
         case SYSCALL_EXCEPTION:
            RTOS->syscall(&QEMU->env); break;
         ...
         }
      }
   } while (true);
}
```

Fig. 2. Mixed level simulation by QEMU task wrapper.

For mixed level cosimulation, we combine our in-house SystemC aRTOS library with the QEMU emulator at the system call interface. The abstract RTOS model abstracts the kernel space whereas QEMU emulates user space on an instruction and register accurate CPU abstraction. For this, each software task is wrapped into its own QEMU user mode emulator under control of an *SC_THREAD*.

Since QEMU has no notion of an execution time, we use a dynamic estimation approach for cycle-approximation during binary translation. Execution time estimation of each task is considered by the SystemC aRTOS model as a delay annotation for its representing SC_THREAD. Then, the QEMU execution is synchronized along the RTOS scheduling policy. In order to reduce the synchronization overhead due to cosimulation, context switching and task preemption is abstracted in the aRTOS model by means of an efficient synchronization scheme that still provides precise interrupt simulation.

Figure 2 depicts our mixed level cosimulation environment with non-native tasks connected with native tasks together for a common RTOS model in SystemC. For this, each non-native task is wrapped by a QEMU wrapper module providing interfaces to the RTOS model and SystemC HW models. The QEMU wrapper provides the synchronization with the RTOS model by means of task execution control, system call handling, and I/O via TLM-interfaced HW models. The core of the QEMU wrapper is the SC_THREAD that controls the main execution loop of a QEMU CPU. Execution delays are estimated during binary translation and used for the synchronization by calling the RTOS model $sync()$ function invoking a call to the aRTOS $CONSUME\_CPU\_TIME()$ function to define a task's execution time.

3.1 Dynamic Binary Translation

A suitable simulation model always has to consider the trade-off between accuracy and performance. Sometimes fast simulation is more important than cycle-true accuracy. This is especially during the early design phase of a complex system with a bunch of CPUs. Thus, we postpone the use of traditional cycle-accurate CPU models to a later refinement step since their use results in a slowdown in runtime by several orders of magnitude. Instead, we use QEMU as an instruction and register accurate abstraction of the target CPU to trade-off some accuracy for performance gain.

QEMU is a software emulator which is based on a dynamic binary translation for efficient conversion of a target Instruction Set Architecture (ISA) into a host ISA with the support of multiple platforms, e.g., ARM, PowerPC, MIPS, or Microblaze. The effort of porting QEMU to new target and host platforms is minimized by means of mapping instructions to an intermediate code, i.e., a canonical set of micro operations.

For the binary translation, target code is considered on Basic Block (BB) level, i.e., linear code segments until a final branch instruction. QEMU uses a dynamic code generator to translate BBs at run-time by means of concatenating precompiled host code segments. For faster execution, Translated Basic Blocks (TB) are stored in a TB cache. Then, the major translation effort is just chaining TBs from cache and patching the instruction operands.

Dynamic binary translation is widely used in a variety of hardware virtualization tools, e.g., Bochs or Sun's VirtualBox. However, QEMU supersedes them and combines some unique features that makes it particularly applicable for our purpose. In general, QEMU can operate in two system modes: user mode and full system mode. The user mode supports user space emulation of a single task on top of a Linux process. The full system mode includes an entire target platform with I/O and kernel space for an operating system and driver execution.

3.2 Dynamic Execution Time Estimation

Instruction Set Simulators are widely used to estimate target SW performance in a virtual prototyping environment. ISS can be either cycle-accurate by using CPU models on Register Transfer Level (RTL) or they can be instruction accurate by using an interpretive simulator. In contrast, QEMU uses binary translation avoiding cycle-accurate CPU models and interpretive execution.

As we can reach significantly higher simulation speeds with the binary translation compared to cycle-accurate ISS models, we do not aim at a cycle-true accuracy. Since QEMU does not provide execution times for the executed code, following the concepts of [10], we extended QEMU by an efficient dynamic estimation approach for cycle-approximate timed execution.

The estimation approach is tightly related to the binary translation. It comprises two levels. In the first phase, each time QEMU encounters a new BB, a static timing analysis of the target code is performed. For this, cycle count values are accumulated for each BB during binary translation. Cycle count values per instruction can be derived from either the CPU specification or by means of estimating average values. In order to reduce the error of dynamic instruction delays, e.g., due to branch misprediction or cache misses, special code can be inserted at the cost of an increased overhead to resolve the error during execution by means of accumulating some extra amount of cycles.

Obviously, the accuracy and efficiency of the estimation approach depends on the complexity of the target platform. In order to achieve predictable systems, most embedded platforms use simple RISC CPUs and avoid multi-staged pipelines with caches. Thus, accumulating a static amount of cycles per instruction is a reasonable abstraction to avoid complex and time-consuming cycle-accurate CPU models.

3.3 Inter-task Communication and I/O

Modern CPUs with RTOS provide several ways for tasks to communicate with their environment, e.g., inter-task communication via kernel primitives or shared memory access. I/O devices can be accessed via memory mapping, i.e., I/O registers connected to a CPU bus are mapped into the CPU address space. Some CPUs also provide special operations for direct access to I/O ports.

Our cosimulation environment supports mixed level task simulation, i.e., cosimulated ISS tasks coexecute together with native tasks on top of a SystemC aRTOS model. For task communication support, we need to define data exchange interfaces between the different simulation models. Exchanging data via kernel space primitives is covered by the syscall translator since each kernel communication invokes a call to the system call interface. Here, we catch system calls from user mode that are passed through by the QEMU by translating the Application Binary Interface (ABI) specific calls from user mode emulation and mapping them to the aRTOS model API.

Communication via shared memory is either provided at compile-time or dynamically using a kernel API. The memory access itself is performed via simple memory operations in user mode. Thus, we must provide a mechanism to redirect shared memory access to a common memory model in order to synchronize shared data between SystemC tasks and QEMU tasks. QEMU provides an API to add memory-mapped I/O

(MMIO) in order to interface HW models written in C. This feature allows to include shared memory models that can be accessed from within QEMU and also from SystemC.

Task I/O can be realized in a similar way. For MMIO, hardware models can be connected via a memory-mapped TLM transactor using the same mechanism as for shared memory simulation. Communication via I/O port accessing CPU operations can be caught and redirected to the TLM transactor via function calls that are inserted during the binary code translation.

3.4 Synchronization and Task Preemption

Precise simulation of interrupts and task preemption is crucial for a sufficient accuracy with respect to the sequence of data accesses and response times analysis. The simulation of an aRTOS model and user space emulation need to be synchronized from time to time since this is the only possibility to yield over control to the RTOS kernel thus allowing the scheduler to preempt task execution. Hence, task preemption granularity is tightly related to the synchronization scheme used by the cosimulation.

Since lock-step synchronization is usually a major reason for performance decrease in cosimulation, we partition the software code of a task into more coarse-grained segments for their timing annotation. For this, we apply our causality-true preemption scheme[4] that is comparable to the approach described in [17]. It abstracts the real preemptive behavior by enforcing a synchronization only for a task interaction, e.g., system calls, I/O, and shared memory access. Thus, the functional execution of a task can only be preempted at interaction points. The estimated execution time between two interaction points is considered in the RTOS model by means of a cycle delay annotation.

This is extremely efficient in combination with the dynamic estimation approach described in Section 3.2 since multiple TBs can be comprised in one execution segment and TB time estimations can be accumulated to a single delay annotation. In the case that all task interaction can be detected during simulation, the application of our synchronization scheme does not influence the causal order in their execution.

QEMU supports two execution modes. In single-step mode, QEMU returns from its execution loop after each target instruction to check whether there are pending interrupts to be handled. In default mode, interrupts are asynchronously triggered and QEMU checks for interrupts only at TB level. This is much more efficient due to the internal TB caching. Along our synchronization scheme, interrupts must be checked before task interaction. Thus, the BB translation must be extended in order to ensure interaction points to be always on the border of a TB. For this, we modified QEMU to finish a TB not only at branch instructions (which is the common understanding of a basic block) but also when an interaction is detected.

Figure 3 compares the different levels of functional segmentation for delay annotated simulation including our modified TB level denoted as *TB**. The difference between BB and TB* levels is that there are additional cuts whenever there is a task communication. These cuts refer to a synchronization between the ISS and the RTOS model so that all TBs in between are comprised to a single execution segment.

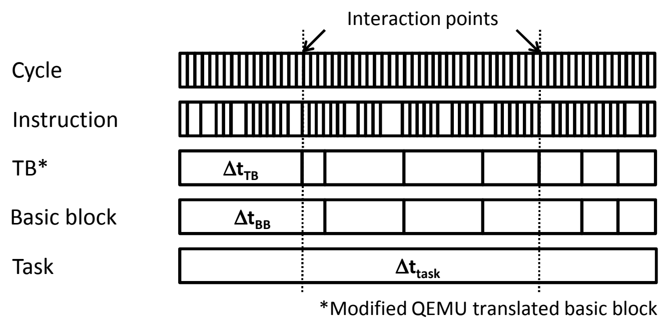


Fig. 3. Comparison of functional segmentation levels.

Basically, interaction detection is covered by the task communication interfaces described in Section 3.3. Nevertheless, BB translation cannot distinguish between memory operations referring to a memory register and those referring to a I/O register since it depends on the instruction operands that are updated during TB chaining. Therefore, as our modified BB translator has no detailed knowledge about I/O, it pessimistically cuts a TB at each memory operation in order to maintain full interrupt accuracy. This may result in a performance decrease due to less efficient TB caching. However, in system level design the system communication is explicitly modeled and refined in a top-down strategy. As such I/O can be derived from a more abstract model to expose potential I/O operations to the BB translation (see Figure 4).

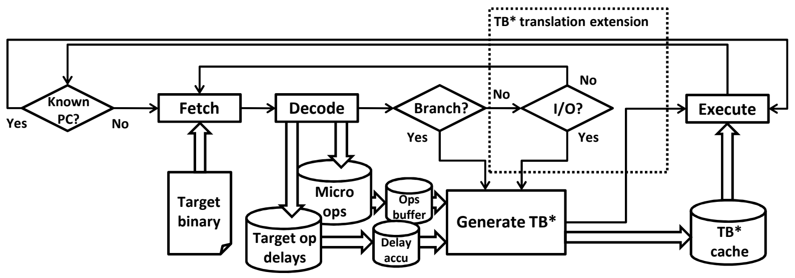


Fig. 4. Modified basic block translation (adapted from [10]).

In order to obtain accurate output for scheduling analysis, the aRTOS model uses an interruptible wait algorithm to hide the abstracted preemption behavior. For this, the delay annotation of a segment is split according to the scheduling policy considering the true preemption behavior on a real target, i.e., true number of preemptions, true preemption point in time, and also the true context switching overhead.

4 Experimental Results

In order to show the feasibility and efficiency of our approach, we applied our simulation environment to the stepwise refinement of an example composed of two computation intensive software tasks running at 100% utilization of the processor. Task 1 iteratively computes prime numbers. Task 2 recursively computes faculties of *n*. The task set is scheduled by a fixed priority scheduler and their execution is synchronized through kernel signals.

Along the refinement introduced in Section 1, we executed several experiments (see Table 1). The first experiment executes the example on top of a native SystemC simulation using our SystemC RTOS library aRTOS. For this, the application C code is wrapped by *SC_THREADs*. For functional segmentation and time annotation, the code is instrumented with preprocessor macros at the branch level. Thus, a task's execution time is considered by means of dynamically accumulating back annotated delays during simulation. According to our synchronization scheme a task yields control to the RTOS model at interaction points. For this, some special macros additionally invoke the function which defines the time delay (*CONSUME_CPU_TIME*) which also enforces a synchronization with aRTOS and the SystemC kernel.

The next experiment runs the software tasks in a mixed level cosimulation according to the mixed level environment introduced in Chapter 3. For this, Task 2 is cross compiled for a PowerPC405 to be executed in our QEMU user mode wrapper. Task 1 remains as a native SystemC thread. At this level the mixed level task set is scheduled by the common aRTOS scheduler in SystemC. In the next step, both tasks are executed in their own QEMU user mode wrapper. The RTOS kernel is still abstract and tasks are scheduled by aRTOS. On the next level, we completely replace aRTOS by full system mode QEMU in order to introduce the actual RTOS kernel. For this, we took our in-house real-time operating system ORCOS[2].

So far, all levels apply our causality-true preemption scheme synchronizing at interaction points since it allows us to operate QEMU in the very efficient TB execution mode in order to take full advantage of the binary translation. In system mode, we switch to single instruction mode in order to achieve full interrupt accuracy. At this level, we do not cosimulate any hardware models except the ones that are provided by the QEMU full system emulator. Finally, we coupled the QEMU system emulator with a SystemC kernel for the cosimulation with SystemC HW models. For this, we synchronize with the SystemC kernel after each emulated target instruction by a SystemC $wait()$ statement.

The experiments were performed on an Intel Core 2 Quad CPU @ 2.4 GHz equipped with 6 GB memory. The target code was built with the GCC 4.4 PowerPC EABI cross compiler. We adopted the QEMU user mode and system mode emulators from the QEMU 0.12.1 release. Each application task was activated 1.000.000 times. The experiments were compared by means of the task output and simulation overhead. Our experiments showed that the task output was equal for all refinement levels thus proving a functional correct simulation. Figure 5 depicts the simulation overhead which we have measured at the different refinement levels.

The plain aRTOS SystemC model required the lowest overhead with just 5.6 seconds for all task activations. Surprisingly, it turned out that migrating the application

Level	Description	Sim. time
aRTOS	All tasks@SystemC w. aRTOS	5.6s
Mixed tasks	Task1@SystemC/Task2@QEMU user mode w. aRTOS	7.6s
Qemu user	All tasks@QEMU user mode w. aRTOS	9.2s
Qemu system	ORCOS@QEMU full system mode	51.6s
QS cosim.	ORCOS@QEMU full system mode w. SystemC cosim.	1472.2s

Table 1. Experiments according to the refinement of the example application.

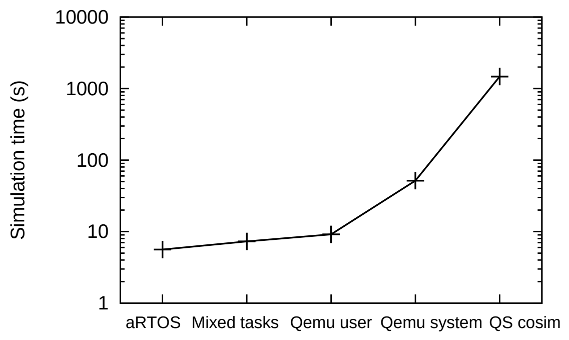


Fig. 5. Simulation overhead increase at different refinement levels.

tasks from aRTOS to QEMU user mode, resulted in a slowdown of just 1.5x for one of the tasks being executed on QEMU and 2x for both tasks, respectively. This is extremely fast since traditional ISS approaches usually come with a slowdown of 4000x-40000x [19] compared to native executed simulation C code. Our experiments also showed that the slowdown of the QEMU execution is nearly linear to the utilization produced by the task set moved to user mode emulation. Thus, using our mixed simulation environment the simulation effort can be considerably reduced when simulating just a subset of tasks on QEMU.

Furthermore, the experiments showed that our mixed level simulation reaches a performance gain of 5.5-6.8x compared to the execution of ORCOS on the QEMU full system emulator and 150-200x compared to the cosimulation of QEMU full system emulator with SystemC, respectively. However, since CPU idle times are abstracted by aRTOS, we expect the performance gain to be even higher with a utilization of less than 100% which is the typical case in hard real-time scheduling. For instance, the worst-case CPU utilization of a task set scheduled by Rate Monotonic must not exceed 69% in order to be feasible [6]. Thus, the average utilization is typically lower.

5 Conclusion

In this article, we presented an approach for the stepwise RTOS-aware refinement of software tasks by means of a mixed level simulation combining the native speed of an abstract SystemC RTOS model and the advantage of the QEMU software emulator. For mixed level simulation, each task can be moved between host-compiled code and cross-compiled target code. Our experimental results show that user mode QEMU with integrated abstract RTOS simulation is a most efficient intermediate step for the migration of native SystemC models to full system emulation and ISS, respectively.

In this context, we can take full advantage of QEMU's efficient binary translation in combination with abstract RTOS simulation. As it makes simulation by several magnitudes faster than traditional ISS, this technology is well applicable for early design phases. Additionally, it provides an ideal intermediate refinement level to smoothly migrate from the introduction of abstract RTOS calls to full system calls of the target RTOS or OS, respectively. Future work will focus on further investigation for the correspondence of the system calls between those levels to increase the automation of the refinement.

Acknowledgments

The work described herein is partly funded by the German Ministry for Education and Research (BMBF) through the SANITAS (01M3088) and the VERDE projects (01S09012) and by the EU through the COCONUT project (FP7-ICT-3217069).

References

1. Homepage of SystemC: http://www.systemc.org/.
2. Homepage of ORCOS: https://orcos.cs.uni-paderborn.de/orcos/.
3. M. Becker, G. Di Guglielmo, F. Fummi, W. Mueller, G. Pravadelli, and T. Xie. Rtos-aware refinement for tlm2.0-based hw/sw designs. In *DATE '10: Proceedings of the conference on Design, automation and test in Europe*, 2010.
4. M. Becker, H. Zabel, and W. Müller. Integration astrakter RTOS-Simulation in den Entwurf eingebetteter automobiler E/E-systeme. In *MBMV'09: Proceedings of the 12th Workshop of Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen*, March 2008.
5. Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.
6. G. C. Buttazzo and G. Buttanzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Kluwer Academic Publishers, Norwell, MA, USA, 1997.
7. P. Destro, F. Fummi, and G. Pravadelli. A smooth refinement flow for co-designing hw and sw threads. In *DATE'07: Proceedings of Design, Automation and Test in Europe*, New York, NY, USA, 2007. IEEE Computer Society.
8. D. Gajski, J. Zhu, R. Dömer, A. Gerstlauer, and S. Zhao. *SpecC: Specification Language and Methology*. Kluwer Academic Publishers, Norwell, MA, USA, 2000.
9. A. Gerstlauer, H. Yu, and D. Gajski. RTOS Modeling for System Level Design. In *DATE'03: Design, Automation and Test in Europe*, 2003.
10. Marius Gligor, Nicolas Fournel, and Frédéric Pétrot. Using binary translation in event driven simulation for fast and flexible mpsoc simulation. In *CODES+ISSS '09: Proceedings of the 7th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, pages 71–80, New York, NY, USA, 2009. ACM.

11. H.M. AbdElSalam Hassan, K. Sakanushi, Y. Takeuchi, and M. Imai. RTK-Spec TRON: A Simulation Model of an ITRON Based RTOS Kernel in SystemC. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pages 554–559, Washington, DC, USA, 2005. IEEE Computer Society.
12. M. Krause, O. Brinkmann, and W. Rosenstiel. A SystemC-based Software and Communication Refinement Framework for Distributed Embedded Systems, 2006.
13. Matthias Krause, Dominik Englert, Oliver Bringmann, and Wolfgang Rosenstiel. Combination of instruction set simulation and abstract rtos model execution for fast and accurate target software evaluation. In *CODES/ISSS '08: Proceedings of the 6th IEEE/ACM/IFIP international conference on Hardware/Software codesign and system synthesis*, pages 143–148, New York, NY, USA, 2008. ACM.
14. M. Monton, A. Portero, M. Moreno, B. Martinez, and J. Carrabina. Mixed SW/SystemC SoC Emulation Framework. 2007.
15. H. Posadas, J. A. Adamez, E. Villar, F. Blasco, and F. Escuder. RTOS modeling in SystemC for real-time embedded SW simulation: A POSIX model. *Design Automation for Embedded Systems*, 10(4):209–227, December 2005.
16. G. Schirner and R. Dömer. Introducing preemptive scheduling in abstract rtos models using result oriented modeling. In *DATE '08: Proceedings of Design, Automation and Test in Europe*, New York, NY, USA, 2008. IEEE Computer Society.
17. Meng-Huan Wu, Wen-Chuan Lee, Chen-Yu Chuang, and Ren-Song Tsay. Automatic generation of software tlm in multiple abstraction layers for efficient hw/sw co-simulation. In *DATE '10: Proceedings of the conference on Design, automation and test in Europe*, 2010.
18. H. Yu. *Software Synthesis for System-on-Chip*. PhD thesis, University of California, Irvine, 2005.
19. Henning Zabel, Wolfgang Mueller, and Andreas Gerstlauer. Accurate RTOS Modeling and Analysis with SystemC. In *Hardware-dependent Software*, pages 233–260, 2009.