RTOS-Aware Refinement for TLM2.0-based HW/SW Designs

Markus Becker†, Giuseppe Di Guglielmo†, Franco Fummi†, Wolfgang Mueller†, Graziano Pravadelli†, Tao Xie‡
†C-LAB, University of Paderborn, Germany
‡Dipartimento di Informatica, Università di Verona, Italy
{firstname.lastname}@c-lab.de
{firstname.lastname}@univr.it

Abstract

Refinement of untimed TLM models into a timed HW/SW platform is a step by step design process which is a trade-off between timing accuracy of the used models and correct estimation of the final timing performance. The use of an RTOS on the target platform is mandatory in the case real-time properties must be guaranteed. Thus, the question is when the RTOS must be introduced in this step by step refinement process. This paper proposes a four-level RTOS-aware refinement methodology that, starting from an untimed TLM SystemC description of the whole system, progressively introduce HW/SW partitioning, timing, device driver and RTOS functionalities, till to obtain an accurate model of the final platform, where SW tasks run upon an RTOS hosted by QEMU and HW components are modeled by cycle accurate TLM descriptions. Each refinement level allows the designer to estimate more and more accurate timing properties, thus anticipating design decisions without being constrained to leave timing analysis to the final step of the refinement. The effectiveness of the methodology has been evaluated in the design of two complex platforms.

1. Introduction

To manage the complexity of designs for refinement and synthesis, Gajski’s Y-Chart [1] were introduced several years ago. At that point in time, the Y-Chart was mainly applied for Register-Transfer and Gate Level synthesis. However, it already covered a so-called system level that was not really well investigated at that time. With the outcome of SystemC, Transaction Level Models (TLM) were introduced in combination with PV (Programmer’s View), PV-T (PV with Time), and CA (Cycle Accurate) refinement. Therefore, most recently, in the context of the TLM 2.0 standard development, time-annotated models were divided into loosely time (LT) and approximate time (AT).

Most of the methodologies still focus on HW and largely ignore the role of SW and the operating system or real-time operating system (RTOS), respectively. In the context of SpecC and the ESE framework, Abdi et al. [2], for instance, elaborate on Task Level abstraction, which introduces an abstract RTOS model as well as processes and communication channels, which are refined into RTOS tasks. In the processor models at the Firmware (FW) and TLM refinement steps, hardware abstraction (HAL) and processor hardware layers are introduced. FW and TLM levels add models of the external bus communication (drivers and bus interfaces) and the interrupt handling chain (interrupt handlers and interrupt logic including processor suspension) on the software and hardware side, respectively.

These levels are mainly introduced for additional abstraction of the OS and the processor as traditional Instruction Set Simulation becomes infeasible with increasing number of processors. Though there is a trade-off between speed and accuracy of models, considerable accuracies could be reached, with an error of less than a few percentage with a simulation speed increase of approximately 1000x at the same time compared to traditional ISS models. However, though there are meanwhile some methodologies considering the combined HW/SW modeling and simulation, they either lack detailed development steps for practical application or do not consider timing information for the real application of TLM 2.0.

This paper introduces a methodology for RTOS-aware task to transaction level refinement which is based on the latest TLM 2.0 standard. For SystemC-based refinement it combines latest automation technologies like processor and RTOS abstraction for fast simulation and seamless RTOS synthesis from the the abstract RTOS to the integration of the actual RTOS/OS via the QEMU virtual machine for true HW/SW co-simulation. Main advantage of this approach concerns the evaluation of timing properties from the early design refinement steps. In fact, as shown in the experimental results, timing properties evaluation at the ISS level becomes infeasible due to the high computational time. On the contrary, the proposed RTOS-aware refinement methodology allows to take design decisions, concerning timing, by simulating very fast abstracted views of the final design.

The remainder of this paper is structured as followed. The next section discusses related works. Section 3 presents the actual methodology before Section 4 gives some experimental results for the refinement of two examples. Finally, this article closes with a summary and conclusions.

2. Related Works

Related works concerning this paper can be organized in three categories: techniques addressing simulation of HW and SW components, methodologies proposing HW/SW refinement flows, and RTOS abstraction approaches.
2.1. Simulation of HW and SW Components

Concerning SW simulation, some solutions are based on the presence of an instruction set simulator (ISS) [3], which is specifically designed to emulate the target CPU. Traditional ISS tools read target instructions and jump to a functionally equivalent code [4]. Schnerr et al. in [5] investigated the use of binary translation for cycle accurate System on Chip (SoC) prototyping.

Another approach is based on the dynamic translation of target code into the host code as performed by QEMU [6]. QEMU is a fast and portable emulator for many architectures (X86, ARM, SPARC etc.) which runs on several architectures. It provides fast simulation performance and it approaches real-time when the target and the host CPUs are of the same type.

Among ISS-free schemes, a delay-annotated software simulation tool is described in [7]. However, this work lacks detailed information on how to determine the delay of each C++ statement. Moreover, it does not consider compiler optimizations. Delay annotation has been improved by considering the Assembly instructions generated by the actual compiler for the target platform in [8, 9]. Moreover, in [8] delay annotation has been combined with native execution of target code on the host platform. Cycle-approximate performance estimation technique for automatically generated TLM models has been presented in [10].

Concerning the simulation of the HW part, a possible solution is the creation of virtual prototypes through the use of HW description languages. Such virtual prototypes are available at the early stage of the design flow and the designer can use different abstraction levels trading off between accuracy and speed. Another solution emulates HW by using inexpensive FPGA [11], but the HW description has to be at RT level to be used on the FPGA.

2.2. Refinement of HW and SW Components

Refinement of HW and SW components has been historically connected with the concept of HW/SW co-simulation and co-design [12, 13, 14, 15, 16, 17]. Early co-simulation approaches require to set up complex heterogeneous environments where HW and SW parts are executed by using different simulators [18, 19]. This heterogeneous style is sub-optimal in terms of simulation performance and easiness of integration.

On the contrary, homogeneous environments use a single engine for the simulation of both HW and SW components, thus simplifying the design modeling. The Ptolemy [12], Polis [13] and Statemate [20] environments are pioneering works in that direction. However, they are suitable only in a very initial phase of the design, since they are based on formal models of computation that do not provide designers with support for easily moving towards the commercial RTL and gate-level synthesis tools generally adopted in the HW refinement flow.

To overcome such a limitation, more recent approaches use system level description languages (SLDLs) such as SpecC [21] or SystemC [22], that can be adopted throughout the refinement steps from system level to RTL.

However, although SystemC TLM 2.0 aims at aiding HW/SW integration [23], it still offers little support for the embedded system designer, who wants to include the dynamic real-time behavior, typical of embedded SW and RTOS, in the system model. In particular, SystemC lacks the necessary constructs to provide SW refinement and it does not support thread priority assignment. This represents a strong limitation, considered that embedded software now routinely accounts for 80% of embedded system development costs [24].

Some works [14, 25, 26, 27, 28, 29] have been proposed to overcome the previous limitation by integrating RTOS capability in the C++/SystemC design flow. However, the approach proposed in [14] requires to use a proprietary simulation engine and it needs manual refinement to get the SW code. In [25], no methods for embedded SW code generation are described. In [26] the method imposes code modification when SystemC construct not supported are used in the original description. In [27], a method is proposed to refine an SLDL specification into C code only after HW/SW partitioning.

To facilitate HW/SW partitioning, in [28], the authors propose a SystemC refinement methodology that focuses on using SW abstraction levels to enhance high-level embedded SW modeling support. However, the high flexibility of such an approach is paid in term of simulation time overhead. On the contrary, the refinement flow proposed in [29] allows to perform HW/SW partitioning in a smooth way by replacing the SystemC kernel with a library relying on the underlying operating system. However, it works for RTL descriptions, it does not show how RTOS parameters can be configured, and it is not timing aware.

2.3. RTOS Abstraction

Abstract RTOS simulation at the task level is typically based on partitioning of the application into hardware components and software tasks, including Interrupt Service Routines (ISR). Tasks and ISRs are then further divided into a sequence of time-annotated software segments. Timing information can be back-annotated into each segment from performance measurements or timing estimations obtained through ISS or WCET/WCRT analysis, respectively. In order to accurately capture data-dependent delays, segments are usually defined at the basic block level, though it is possible to partition into more coarse-grain segments.

In [30, 31] C code is transformed to a platform-independent intermediate format that already comprises structural code optimizations. The intermediate format is instrumented in order to abstract the target-dependent assembler code. On the contrary Hastano et al. [32] use a stochastic approach to model the task execution times based on BCET/WCET estimation by means of the Gumble distribution.

Fast RTOS simulation with time annotated segments is either based on standard RTOS APIs or on the implemen-
3. Methodology for RTOS-Aware TLM 2.0 Refinement

Our methodology aims at simplifying system partitioning and HW/SW refinement with focus on the RTOS-aware transition from untimed to timed model and further on to true HW/SW cosimulation in the context of the TLM 2.0 OSCI standard (cf. Figure 1). It is composed of four levels that, starting from an untimed TLM SystemC description of the whole system, progressively introduce HW/SW partitioning, timing, device driver and RTOS functionality, arriving at a final platform, where SW tasks run upon an RTOS hosted by the QEMU SW emulator and HW components execute under SystemC as cycle accurate TLM models.

3.1. Level 1

At Level 1, the system is modeled as an untimed SystemC description for a rapid proof of functional concepts by fast simulation without taking care of lower-level details and timing. For this, Level 1 starts with an unpartitioned HW/SW model as a set of concurrently executed (untimed) C/C++ functions synchronized by events and communicating through channels and shared memory. Processes, functions, and channels represent abstractions of tasks and interprocess communication (IPC) primitives. At this level, a rough abstraction of the RTOS/OS scheduling can be realized by the simulation scheduler. Nevertheless, it is also possible to introduce a first explicit RTOS/OS abstraction which can be later refined to the canonical RTOS API. In a first refinement, we have to identify the main communication structure by introducing one or more explicit communication buses, i.e., replacing simple or complex channels by explicit TLM 2.0 blocking transport interfaces. In order to be TLM 2.0 compliant, we apply the loosely time (LT) scheme and set time parameters to zero in a first step.

3.2. Level 2

For Level 2, the model is refined into an untimed TLM 2.0 SystemC description. Based on standard SystemC syntax, different HW/SW partitioning alternatives can be analyzed very quickly without requiring manual conversion from SystemC code to C/C++ code and vice versa. However, at this level, the model lacks the CPU and the bus descriptions and SW tasks are not synchronized yet.

The different HW/SW partitioning alternatives are rapidly analyzed by the application of our library which replaces the SystemC kernel by a Posix-like kernel to allow...
• SW tasks running as Posix threads upon a Posix-like operating system (e.g., Linux);
• HW components preserving the SystemC simulation Semantics by running upon a dedicated so-called HW manager;

while the syntax of both HW and SW parts remain unchanged, i.e., untimed TLM 2.0 SystemC.

The first benefit deriving from substituting the SystemC simulation kernel with our Posix-based library is that manually modifying modules from HW to SW side and vice versa does not require error-prone and time consuming conversion of SystemC code into C/C++ programs and vice versa. Then, a second benefit derive from the fact that the co-routine execution model\(^1\) of the SystemC simulation kernel is removed. Thus, SW threads can be concurrently executed, possibly on a multi-processor system. Concurrent execution is a mandatory condition for addressing SW refinement issues like mutual exclusion, deadlock avoidance, priority assignment for thread scheduling, etc.

In contrast to the approach proposed in [29], which is mainly targeted for SystemC RTL code, our Posix-based library supports all the coding styles of SystemC TLM 2.0.

3.3. Level 3

For Level 3, we adopt the TLM 2.0 loosely timed (LT) coding style and further refine to approximately timed (AT) models and apply a time-bounded interrupt and scheduling analysis by fast time-accurate simulation and introduce an abstract RTOS layer. For this, we replace the Posix-like kernel back by the standard SystemC kernel and apply an abstract canonical RTOS implementation. For our simulation we have taken the aRTOS SystemC library [35] though comparable libraries would serve the same purpose. aRTOS introduces additional SystemC keywords for the previously identified Level 2 SW tasks and Interrupt Service Routines. At Level 2, the partitioned HW/SW model is available in standard SystemC syntax. Along the lines of aRTOS, for Level 3 refinement, SC_MODULEs with SW task have to be simply renamed as RTOS\_MODULEs which are associated to a RTOS\_context for which a standard task scheduler has to be defined, i.e., parameter of an existing scheduler have to be determined or a new scheduler has to be implemented. After that, processes which are defined inside the RTOS\_MODULE are determined as software tasks which are scheduled by the respective task scheduler. Additional keywords identify ISRs and implement interrupt management. Correspondingly, an interrupt scheduler has to be implemented or selected. Multicore platforms can be easily modeled by the introduction of multiple RTOS contexts where each may implement different SW and interrupt schedulers.

In a further step, tasks and ISRs are divided into a sequence of time-annotated software segments. Timing information are back-annotated into each segment from performance measurements or timing estimations obtained through ISS or WCET/WCRT analysis, respectively. In order to accurately capture data-dependent delays, segments are usually defined at the level of basic blocks though it is possible to partition them into more coarse-grain segments.

The syntactical replacement of SC\_MODULEs by RTOS\_MODULEs at this level provides the basis for further separation of HW and SW for true co-simulation on the next level, where the final SW runs on the actual RTOS/OS that is installed and running on the QEMU virtual machine, which in turn is linked with the SystemC model for the HW.

3.4. Level 4

Finally, for Level 4 HW components and the bus model are refined to the cycle accurate TLMs, while SW binaries are executed upon the real target RTOS. Level 4 relies on definition of a co-simulation framework where cycle accurate TLM SystemC is used to model the HW, and the QEMU SW emulator to execute RTOS and SW based for a specific instruction set architecture like PowerPC 405.

For co-simulation, QEMU is modified to communicate with the SystemC simulation kernel to set up the final HW/SW cosimulation framework. Such a framework is completely transparent to SW and HW engineers, which can focus, respectively, on SW engineering, RTOS configuration and device driver generation, and on HW refinement. Performance analysis is timing accurate and completely aware of RTOS parameters.

The co-simulation framework allows SW application to access HW components as it happens in actual embedded platforms, i.e., through device drivers that send information via memory-mapped registers and device memory. Registers and ports are mapped as memory locations (I/O memory). This way, they are available to the processor over the bus at precise addresses. Operations on those addresses are recognized as operations on the device registers, and thus they are handled by the device itself.

In particular, device drivers read and write the device registers through the I/O memory slots where the device is registered. Communication between QEMU and SystemC is established by an interprocess channel (socket-based communication) and ISS ports. The ISS ports have been added to the SystemC library as an extension of standard sc\_in and sc\_out ports. This mechanism substitutes the direct access to I/O mapped memory of real operative systems with real HW devices. The link between SystemC ports and memory addresses in the QEMU is implemented by using a binding table stored in the SystemC kernel. Therefore, implementing such a co-simulation required:

• modifying QEMU both to communicate with the SystemC simulator and to manage the HW device;
• modifying the SystemC simulator kernel to add the capability of reading and interpreting the messages coming from the QEMU side, as well as sending interrupts
to QEMU whenever the HW models generate them. These operations are transparent to the designers who only need to write the model by using the standard SystemC statements.

Device driver can be either manually defined or automatically generated according to the flow proposed in [37].

4. Experimental Results

Two industrial platforms have been considered to validate the proposed refinement methodology, ERA and MAGALI OFDM.

The ERA platform has been provided by STMicroelectronics. The platform is composed of three IP modules, a memory, and a testbench interconnected by an AMBA AHB interface protocol as depicted in Figure 2. In particular, the Adaptive DPCM (ADPCM) is a variant of DPCM (differential pulse-code modulation) that varies the size of the quantization step, to allow further reduction of the required bandwidth for a given signal-to-noise ratio in voice over IP systems. The ECC is used for single-/multiple-bit(s) error detection/correction in read/write operations on memory elements to improve the reliability and functioning of the system. Finally, the ROOT module supplies square root, inverse square root, and some other elementary functions used by the ECC module.

The MAGALI OFDM platform has been provided by CEA-LETI. The orthogonal frequency division multiplexing (OFDM) is a special form of multi-carrier modulation. It is particularly suited for transmission over a dispersive channel. The OFDM refers to the fast Fourier transform (FFT) in the up/downlink flow as depicted in the block diagram in Figure 3.

All experiments have been carried out on an eight-processor Intel Xeon 2.8 MHz equipped with 8 GB RAM and 2.6.23 Linux kernel. Both scenarios have been originally described with the OSCI TLM-2.0 library.

Table 1 reports the number of lines of code of each scenario for the levels of the proposed methodology (L1-L4).

Table 2 details the simulation time, user time (uTime) and kernel time (kTime), at the same levels. For Level 3 results are provided for both TLM Loosely Timed (LT) and TLM Approximately Timed with four phases (AT4) coding styles.

Looking at Table 2 several considerations arise. From L1 to L2 the increase of kTime is definitely due to adding the Posix kernel instead of SystemC kernel. In that way, all the processes run upon the underlying operating systems originating many system calls and many context switches. The increase in uTime is due to the presence of the HW manager, that at the moment, is not as much optimized as the SystemC kernel.

In a similar way, from L1 to L3 LT, increase in uTime runtime is mainly due to an increased synchronization in the SystemC model. Unlike L2, kTime is not affected, since, aRTOS runs with SystemC in user mode. The overhead mainly originates from annotating the SW parts with timing information, i.e., moving the models from untimed to unscheduled timed models. For this we have to annotate the code by \texttt{CONSUME\_CPU\_TIME} statements on basic SW block level. This statement in turn executes SystemC wait statements enforcing a SystemC context switch each.

In the case of ERA, the simulation executes 5,502,554 \texttt{CONSUME\_CPU\_TIME} statements, each of which executes 2.41 SystemC context switches in average. In summary, we could see for ERA that the introduction of aRTOS calls gives a slow-down by a factor 4.3. 3.4 of it is due to the execution of additional SystemC wait statements. The plain aRTOS functionality gives us a 26% overhead on top of it. However, the total factor of 4.3 from L1 to L3 LT cannot be generalized since it depends on the individual model.

In the case of OFDM, we can see from the figures that the overhead for time annotation and RTOS scheduling is much
less and negligible compared to execution of the actual Fast Fourier Transformation (FFT) computation. For both models, the increase in runtime L3 LT to L3 AT4, i.e., is just due to the replaced bus model refining LT to AT4.

Concerning the SW scheduling, on L2 we have applied a Round Robin based scheduler under Posix. Our analysis in L3 has shown that the system performs correctly with this scheduler. Therefore, we also kept Round Robin for L4. For both examples, we could show at L3 that the models also have a correct function with Fixed Priority scheduling. This information is important for later design re-spins, since the latter gives a better performance due to less context switches.

At L4, the co-simulation mechanism has a great impact on the simulation time. Thus, we must try to perform the majority of exploration before arriving at L4, and leave at this level of the methodology only the HW/SW interface tuning after the introduction of the device driver.

5. Concluding Remarks

This paper proposed a four-level RTOS-aware refinement methodology for HW/SW TLM based designs. Each refinement level allows the designer to estimate more and more accurate timing properties, thus anticipating design decisions without being constrained to leave timing analysis to the final step of the refinement. The effectiveness of the methodology has been evaluated by considering two industrial platforms.

References