Fast and Accurate Timed Execution of High Level Embedded Software using HW/SW Interface Simulation Model

Aimen Bouchhima    Sungjoo Yoo    Ahmed Jerraya

System-Level Synthesis Group
TIMA laboratory
46, Av Félix Viallet, 38031 Grenoble, France
e-mail : {aimen.bouchhima, sungjoo.yoo, ahmed.jerraya}@imag.fr

Abstract - In this paper, we propose a methodology to perform early design stage validation of hardware/software (HW/SW) systems using a HW/SW interface simulation model. Given a SW application described at the OS abstraction level and a HW platform described at an arbitrary abstraction level, we aim at providing the adaptation layer, i.e. simulation model of the HW/SW interface, which will enable the timed HW/SW cosimulation of the entire system at an early design stage before the system design is completed. Experimental results show that our approach is easy to use and efficient while providing fast simulation (up to 3 orders of magnitude faster than a HW/SW cosimulation with instruction set simulator, ISS) and accuracy (86% compared with a HW/SW cosimulation with ISS).

1. INTRODUCTION

Raising the abstraction level is widely seen as a solution to bridge the gap between the increasing complexity of SoCs and the low design productivity. However, despite the emergence of system-level modeling as the solution that relieves the designer from low level implementation details, the effective integration of such high level approaches in a real design flow still presents many challenges. In fact, according to the current design practice, there is still a huge gap between the high level description of the system and its actual implementation.

SW designers should wait until the HW platform is completely refined (at least down to RT level) before they start designing the SW part of the system. It is only at the end of this long refinement process that a global validation of the entire design is performed, generally involving a cycle accurate simulation model for the processors (using instruction set simulators) and RTL models for the HW components.

The conventional validation raises at least two problems: (a) late validation in the design cycle and (b) slow validation. In conventional practices, the validation of entire SoC can be performed only when both SW and HW design is completed. Thus, the debug cycle is long since the debugging can cause re-design at high-level, compilation/synthesis, and low-level re-validation of the entire SW and HW. The conventional validation itself is slow since instruction set simulators (ISSs) and RTL HW simulator(s) are involved in the HW/SW cosimulation.

1.1 High Abstraction Levels of SW and HW for Early and Rapid Validation of SoC

To overcome the problems, one solution is to raise abstraction levels of both HW and SW allowing designers to gradually refine their systems and to perform fast validation at each design step. Figure 1 illustrates high abstraction levels in SW and HW. In the figure, the entire system is viewed as the combination of a SW part (denoted with application SW) and a HW part (denoted with HW platform) that interact with each other via a HW/SW interface.

Note that, at the highest abstraction level (functional level) the HW/SW interface between the “SW part” and the “HW part” practically disappears. The lowest levels are ISA (Instruction Set Architecture) level for SW and RTL for HW. They are used to perform conventional HW/SW cosimulation [1].

As an intermediate HW abstraction level, the figure highlights Transaction Level Model or TLM that has recently raised increasing interest in the designers’ community [2].

TLM (which actually includes several sub-levels such as transfer, transaction and message level [14]) aims at capturing the relevant abstraction of communication in HW depending on the designer’s need.

Concerning the SW, there have been parallel efforts to define a high abstraction level that corresponds to an early SW design stage. We define the operating system (OS) level to be the level where the application SW is explicitly designed using a fixed, well defined OS API (application programming interface).

The OS API includes a set of services exported by the OS. Note that, details about the actual implementation of the OS are irrelevant at OS level. Thus, they are abstracted at this level. The OS behavior, however, is of great importance. Note also that we do not impose any constraint on the OS API. It can be the API of a real OS (whether it is commercial, open source, or in-house application specific) or a generic abstract OS API (which might be convenient when no early decision concerning the use of a particular OS is made).

Based on this OS level, designers will be able to refine the application SW early in the design cycle, while being able to perform the validation of the entire system in term of functional and non-functional properties (e.g. real time aspects). In this validation, designers can take into account the effects of operating system including the HW part (that may be at different abstraction levels). The validation enables to investigate and make design decisions on the implementation parameters (e.g. task priorities) and the

Fig. 1: Abstraction levels in HW and SW
architectural decisions (e.g. OS scheduling policy). It is important to note that the results of such validation could also be valuable to the HW designers who will be able to have, for instance, a more accurate estimation of the communication network activity.

1.2 Requirements of Validation at OS Level

For the validation of the entire system with SW at OS level, a simulation model of HW/SW interface is needed. The model allows SW to be described at OS level, and HW to be modeled at an arbitrary level (e.g. functional, TLM or RTL). The simulation model needs to support

- Different SW design scenarios at OS level
- Different abstraction levels in HW
- Simulation accuracy

The simulation model should be flexible enough to support various design scenarios at OS level, e.g. usage of different OS APIs (generic or commercial ones). It needs to also support different HW abstraction levels, e.g. functional, TLM and RTL. The simulation model needs to be timing accurate enough to enable correct validation of the design decisions made at OS level.

1.3 Our Contribution

In this paper, we present a methodology to build HW/SW interface simulation models that satisfy the above mentioned requirements. Figure 2 gives an overview of the proposed methodology. In Figure 2 (a), a SoC architecture is illustrated. It consists of application SW, OS and HW abstraction layer (HAL) as its SW part. The HW part includes CPU, its local architecture, and the rest of system containing the communication network and the other HW components (that include other CPUs and IPs).

Figure 2 (b) shows a conventional HW-SW cosimulation model that includes ISS and HW simulation model (at RTL). A bus functional model (BFM) plays the role of the HW/SW interface simulation model.

In our methodology, we propose a high-level HW-SW cosimulation model as shown in Figure 2 (c). In the proposed model, the application SW is natively executed on the host simulation machine instead of running an instruction set simulator to simulate it. To support this native execution, a HW/SW interface simulation model is provided.

The structure of HW/SW interface simulation model is illustrated in more details in Figure 2 (d). The simulation model acts as an adapter between the application SW and the HW part. It is composed of two layers: one SW layer, called OS model layer, which constitutes the OS simulation model and one HW layer, called device functional layer, which emulates a set of device controllers. Depending on the nature of the HW part and its abstraction level, functionalities of the device functional layer may vary accordingly. However, the interaction between the two layers remains basically the same (i.e. I/O and interrupt).

These two layers enable flexibility to support the requirements in Section 1.2 (different SW design scenarios at OS level and different HW abstraction levels). The OS model layer enables OS level simulation of application SW and the device functional layer enables HW simulation at an arbitrary level. Simulation accuracy at OS level is achieved by (1) annotating the application SW with SW execution delay and (2) ensuring correct handling of asynchronous HW interrupts in the HW/SW interface simulation model.

![Figure 2: Overview of the proposed approach.](image)

The remainder of this paper is organized as follows. Section 2 reviews related work in high level HW/SW modeling. The proposed methodology is presented in Section 3. Section 4 gives experimental results. Section 5 concludes this paper.

2. RELATED WORK

Several system-level description languages such as SystemC [4] and SpecC [5] have emerged as environments to perform high level HW/SW co-design. However, in terms of SW design, they allow the application SW to be described only at functional level. Without a specific simulation model of HW/SW interface, e.g. OS model, they cannot allow to model the HW/SW interface that includes OS, HAL and HW devices.

In conventional embedded software area, OS emulators are known to provide an environment to execute application SW on the top of specific OS API. Projects like CarbonKernel [6] and Xenomai [7] are examples of such environments. In addition, many commercial embedded OS providers support OS emulators (e.g. VxSim [8]). While they are useful to debug SW at very early design stages, these environments remain SW centric. That is, they lack in HW modeling capabilities, e.g. timed HW simulation model.

SoCOS [9] is an example of an OS model that can be simulated with timed HW simulation models. In [15], a method of building OS simulation model is presented. The model can also be simulated in HW/SW cosimulation in SpecC. However, in [9] and [15], the interaction of OS simulation model with the HW part (i.e. I/O, interrupt handling) is not clearly explained.

Madsen et al. [10] propose a method to perform real time scheduling analysis within the SystemC environment. However, their work is limited to a periodic task model and does not take into consideration other OS related aspects such as I/O and interrupt handling.
3. High Level HW/SW Cosimulation Model

Figure 5 shows the structure of the proposed high level HW/SW cosimulation model. It consists of application SW executed natively on the simulation host, HW simulation models and HW/SW interface simulation model. The simulation model of HW/SW interface is composed of two layers: one SW layer called OS model layer and one HW layer, called device model layer, which emulates a set of device controllers. Each of the two layers enables SW simulation at OS level and HW simulation at an arbitrary abstraction level, independently.

To enable accurate OS level simulation of application SW, the simulation model of HW/SW interface needs to model
- SW task scheduling
- Interrupt handling
- I/O operations (i.e. read/write)
- Execution time of software

To model task scheduling, we need a model of SW task and context switch of SW tasks (Section 3.1). Interrupt handling and timed simulation of application SW are related to each other (Section 3.2 and 3.3). I/O modeling needs to properly handle the interaction between HW and SW (Section 3.4).

![Fig. 5: Structure of proposed cosimulation model](image)

3.1 Hierarchical Scheduling of Multi-thread Application SW

When the cosimulation is performed in a host simulation environment such as SystemC, SpecC, etc., to perform multi-tasking of application SW, we have two solutions. One is to map each of application SW tasks on a process in the host simulation environment as in [15]. In this case, the multi-tasking, i.e. context switch of SW tasks, is done by the host simulator kernel. The other is to perform the multi-tasking of application SW tasks without using the scheduling of host simulator kernel. In this case, for the multi-tasking of application SW tasks, we can use user-level multi-threading functionality provided by the host OS, e.g. Unix.

In our methodology, we choose the second solution since, in the first solution, the host simulator kernel can yield significant overhead (due to frequent process scheduling) in simulation runtime when there are many application SW tasks (mapped on a single target processor). Thus, we run the task scheduler model and its application SW tasks in a process managed by the host simulation kernel. For simplicity of explanation, throughout this paper, we use SystemC as our host simulation environment.

From the viewpoint of application SW tasks, the task scheduling is performed in a hierarchical way. First, a SW task is scheduled by a task scheduler model in the OS model. Then, the OS model is scheduled by the host simulator kernel.

This hierarchical scheduling model is illustrated in Figure 6. In this figure, the rectangles represent the processes managed by a host simulator kernel. The host simulation environment is usually a HW simulator, e.g. SystemC. Thus, the task scheduler model and its SW tasks are mapped on a HW process (called SW-mapped HW process) as shown in the figure. The task scheduler model can take any scheduling policies, e.g. preemptive, non-preemptive, static or dynamic priority-based, etc.

![Fig. 6: Hierarchical scheduler concept](image)

From the viewpoint of building task scheduler models, what is important here is how to enable context switch of SW tasks within the SW-mapped HW process in the hierarchical scheduling. To do that, we use user-level multi-threading functionality provided by the underlying host OS, e.g. Unix, Linux, etc. Table 1 shows an example of user-level multi-threading functionality of host OS. In this case, to model the SW task and context switch, we use structure ‘ucontext_t’ and function ‘swapcontext’. For more details of using host OS to simulate multi-thread application SW, refer to [16].

Compared to the conventional usage of user-level multi-threading for the execution of multi-thread application SW [16], our contribution is to enable the execution of multi-thread application SW in the process managed by a HW simulator (see Section 2 and 3).

<table>
<thead>
<tr>
<th>SW task context</th>
<th>ucontext_t</th>
</tr>
</thead>
<tbody>
<tr>
<td>Context switch</td>
<td>swapcontext</td>
</tr>
</tbody>
</table>

Table 1: Modeling task and context switch in UNIX.

3.2 Modeling SW Execution Time

The basic principle in modeling SW execution time is that only a single task can consume the processor resource at an arbitrary time point. To model the SW execution time, we annotate the application SW code with function ‘consume(delay)’, where function ‘consume()’ emulates the consumption of processor resource (details will be given in Section 3.3) and ‘delay’ represents the execution delay of
application SW code section. The delay can be measured on a prototyping environment or estimated [16][17]. When the function consume() returns, the execution of the corresponding application SW code section is considered to be completed.

During the execution of function consume(), when an interrupt arrives at the target processor, the corresponding interrupt service routine (ISR) should be invoked. The ISR can even call the task scheduler model and another SW task can be invoked by the task scheduler model. That is, the currently running task can be preempted.

3.3 Interrupts Modeling

From the viewpoint of OS model, HW interrupts (interrupts from the HW part to the SW part) are modeled as an sc_event called “interrupt” (associated with an identifier). To handle asynchronous interrupts, we rely on the function consume() inside of function wait(delay, interrupt). This function allows waiting for the event “interrupt” with a timeout equal to “delay”. In other words, the function returns either when the specific event has occurred within the waiting delay or at the end of the timeout. Figure 7 shows the basis of the algorithm used to handle HW interrupts.

3.4 I/O Modeling

To enable the correct modeling of I/O operations, we typically need a model of the I/O interface that we call device functional model. This entity has to ensure the correct interaction with the OS side while providing the necessary adaptation of the communication interface (protocol, abstraction level, etc). This functionality is exemplified in Figure 8. In the figure, FIFO_IN_DRV and HS_OUT_DRV are device driver objects. Each class of the objects exports a set of device driver functions (Get and Put) and includes one or more interrupt service routines (Isr) that are registered within the OS kernel in order to react to external events.

Fig. 8: Examples of device driver / device functional models

In the left side of the figure, the SW task needs to communicate with a high level channel (sc_fifo in SystemC) that provides some blocking functions (e.g. blocking read/blocking write). When a blocking function is called, if the channel is not ready to execute them (e.g. fifo full or empty), the execution is blocked thereby blocking the caller of the function. Thus, these blocking functions cannot be directly called by the SW task. Otherwise, the SW-mapped HW process that encapsulates the entire SW execution will be blocked by the blocking function. In such a case, handling of the other interrupts cannot be done. Thus, multi-tasking cannot be modeled correctly.

To enable multi-tasking during I/O operating, the blocking function of communication channel (in this example, blocking read/write from/to sc_fifo channel) needs to be “viewed”, by the SW side, as a non-blocking function. To do that, we have to decouple the SW simulation and the execution of blocking functions of communication channels. To decouple them, our simulation model uses additional HW processes within the device functional model.

Figure 8 (a) shows an example of device functional model drv_process (in line 14). The process is activated (line 4)
via sc_event ‘driver_event’ when the device driver function (in this case, function Get()) of corresponding device object is called by the application SW task. The blocking operation (in this case, blocking read from the sc_fifo channel) is performed in the device functional model (line 16).

Figure 8 (b) exemplifies a more refined communication channel corresponding to a hand-shake protocol. Here the HW process of the device functional model is used to wait on the external event of the ack signal (line 15) and to trigger the corresponding interrupt service routine (line 17) through the hw_interrupt function provided by the OS simulation model.

4. IMPLEMENTATION AND EXPERIMENTS

4.1 OS Model Library

We applied our methodology to implement an OS model library consisting of a set of C++ classes built in SystemC. This object oriented implementation ensures modular and extensible design environment. Figure 9 shows the class hierarchy of our OS model. On the top of the hierarchy, we find a class called ROOT_OS. This class provides the minimal yet sufficient functionalities required to build operating system models. Examples of these functionalities include context creation and switch, basic interrupt handling, low level access to physical ports etc. Each implementation of an OS model will then inherits those basic functionalities to build its specific features.

ROOT_OS class is a simulation model of the HW abstraction layer (HAL) part of an operating system. In our case, this basic class has about 800 lines of C++ code. The context related functionalities necessary to build the OS task scheduler are implemented using POSIX user level multi-threading library as shown in Table 1.

Concerning the OS model library, we implemented two kinds of model. The first corresponds to a generic OS that features a preemptive FIFO based scheduler with basic task management and synchronization. It has about 1300 lines of C++ code and takes roughly two days/man to write it. The second model corresponds to a real OS, eCos operating system [12]. It is a component based OS that has the advantage of clearly separating the HAL from the processor-independent or portable part. Thus, based on our ROOT_OS class, all we had to do was to write a new HAL component for eCos. For the remaining part of eCos, we just reused the same original code (after minor adaptation to fit into our class library structure). This work took less than one week.

4.2 SW Simulation Model

Figure 10 gives a more detailed view of a typical SW_MODULE class declaration. SW_MODULE inherits from my_os_model (line 1) which is already available in the OS model library. The class may include an arbitrary number of SystemC ports of different types (as indicated by the comment on line 5). This will define the HW interface of the SW module. Within the constructor of the class, only one SystemC process of type SC_THREAD is declared (line 9). This SW-mapped HW process, called SW_process, corresponds to the unique process that will encapsulate the execution of both application SW tasks and the OS task scheduler (see Section 3.1).

Note that in our implementation, we choose to totally decouple the application SW code (one that will be used in the final implementation) from the rest of code intended for simulation purpose (SystemC code). In particular the application SW is compiled separately as static library and then linked together with the whole simulation executable.

```cpp
1 class SW_MODULE : public sc_module, my_OS_model
2 {
3   public :
4   ...
5   //SystemC user ports declaration
6   ...
7   SC_CTOR(SW_MODULE)
8   {
9       SC_THREAD(SW_process)
10       //other initializations
11   }
12   ...
13   void SW_process()
14   {
15       kernel_entry();
16     }
17   ...
18   }
```

Fig. 10: Typical declaration of a SW module.

4.3 Experiments

With our OS model library, we performed HW/SW cosimulation of a VDSL design. The system is composed of two SW modules mapped to two ARM7 processors and one HW IP block [13]. The first SW module consists of three tasks and the second consists of five tasks. We run the simulation at three different abstraction levels. The first level corresponds to the functional level where all the SW tasks of
the system are described as parallel processes in SystemC. This functional model is then refined to the OS level using an OS API. Then, we perform HW/SW cosimulation of the system using an OS model for each SW module. Finally, the system is refined to the ISA level and is simulated using cycle accurate ISSs.

In order to compare the simulation results of the three different simulation runs, we kept almost the same code of the application SW tasks (i.e. without any code transformation or optimization). We also used the same (RTL) model of the IP block throughout the different three abstraction levels. We performed delay annotation of application task code in both functional and OS levels. At the OS level, we also annotated the OS models with estimated delays to reflect the actual overhead of the real OS.

Table 2 shows the obtained simulation results of the three cases. The accuracy of a model is defined based on the time needed by a SW task to accomplish a given work. We consider the time obtained by a cycle accurate simulation as a reference one. The mean accuracy of a given model is computed as follows:

\[ \text{accuracy} = \left( 1 - \frac{\sum_{\text{all tasks}} Tca - Te}{\sum_{\text{all tasks}} Tca} \right) \times 100 \]

where Tca is the reference time needed by a task in the cycle accurate model and Te is the time need by the same task in the considered abstraction level.

<table>
<thead>
<tr>
<th>Abstraction Level</th>
<th>Simulation Runtime</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Functional level</td>
<td>22 s</td>
<td>19%</td>
</tr>
<tr>
<td>OS level</td>
<td>25 s</td>
<td>86%</td>
</tr>
<tr>
<td>ISA level</td>
<td>32,562 s</td>
<td>100%</td>
</tr>
</tbody>
</table>

Table 2: VDSL experimental results

The obtained results show that the OS level simulation achieves a considerable speedup compared to the cycle accurate ISS based one (more than three orders of magnitude) while being comparable to the functional level in term of simulation speed. The speed overhead introduced by the OS simulation model is considerably reduced (to less than 15% of the whole simulation runtime) thanks to the efficient implementation of the hierarchical scheduler model (see Section 3.1).

Concerning the accuracy, the functional model gives, as expected, a poor accuracy. In fact, at this level, neither the overhead inherent to the operating system nor the task serialization is taken into consideration. Using the OS level simulation, those aspects are now modeled and the overall simulation accuracy is considerably improved (86% accuracy). The error of OS level simulation is mainly due to the fact that the HW interface of each SW module is modeled at a higher abstraction level than the actual processor interface used at the ISA level. Moreover, the errors of delay estimation contribute also to the total error.

5. CONCLUSION

In this paper, we presented a methodology for high level HW/SW cosimulation that allows building HW/SW interface simulation models including operating system models. This enables to reduce the debug cycle by both early stage validation of the entire HW/SW design and fast validation at a high abstraction level.

We implemented an OS model library with two OS models: one generic OS and the other for eCos. The experiments of a VDSL sub-system showed the proposed methodology enables fast HW/SW cosimulation comparable to the cosimulation with functional level SW simulation and good accuracy (86% compared with the accuracy of HW/SW cosimulation with ISS).

ACKNOWLEDGMENTS

This research has been sponsored in part by the ToolIP-A511, SpeAC-A508 projects of the Europe’s MEDEA+ (Microelectronics for European Applications) program.

REFERENCES