Virtual Hardware Prototyping through Timed Hardware-Software Co-Simulation

Franco Fummi, Mirko Loghi, Stefano Martini, Marco Monguzzi, Giovanni Perbellini, Massimo Poncino

Virtual Hardware Prototyping through Timed Hardware-Software Co-simulation

Franco Fummi[†] franco.fummi@univr.it
Mirko Loghi[†] loghi@sci.univr.it
Stefano Martini[†] martini@sci.univr.it
Marco Monguzzi[#] monguzzi@sitek.it
Giovanni Perbellini[†] perbellini@sci.univr.it
Massimo Poncino[‡] massimo.poncino@polito.it

[†] Dipartimento di Informatica - Universitá di Verona
Strada Le Grazie, 15 37134 Verona, Italy
[‡] Dipartimento di Automatica e Informatica - Politecnico di Torino
Corso Duca degli Abruzzi, 24 10129 Torino, Italy
[#] SITEK S.p.A.
Via Monte Fiorino, 9 37057 S.G. Lupatoto - Verona, Italy

Abstract

Designers of factory automation applications increasingly demand for tools for rapid prototyping of hardware extensions to existing systems and verification of resulting behaviors through hardware and software co-simulation. This work presents a framework for the timing-accurate co-simulation of HDL models and their verification against hardware and software running on an actual embedded device of which only a minimal knowledge of the current design is required.
Experiments on real-life applications show that early architectural and design decisions can be taken by measuring the expected performance on the models realized using the proposed framework.

1. Introduction

Co-simulation allows functional and/or timed verification of mixed hardware-software systems. When a complete hardware (HDL) model of the system is available, a wide range of techniques and tools support designers in prototyping additional hardware and software to extend the system itself. In many situations, however, particularly whenever systems are already on the market, designers may face requests for extending systems that grew over time as merging of different parts, some written with different modeling languages and some others drawn as schematics; This complicates the process of correctly reverse-engineering the project, or requiring different tool versions setup (some of which not more supported anymore, in some cases). Furthermore, designers may not have access to some parts of the project because of intellectual property issues.
In this work, we propose a methodology for (i) extending an hardware system that requires minimal knowledge of the current design and (ii) allows an effective timing-accurate simulation of the new project. Time-to-market may particularly benefit from this approach to rapid prototyping, by increasing the chance of success in a project where new hardware has to extend actual systems. Furthermore, by supporting timing-accurate co-simulation, the framework fulfills the severe timing requirements in verifying software and hardware mixes for industrial control applications.

The proposed methodology explicitly targets the interaction of a microprocessor-based board, which hosts an operating systems and executes the software portion of the design, and a hardware simulator, which models the device under design. In that respect, it sensibly differs from other previous approaches, since the co-simulation must deal with issues other than just the (timing-accurate) interaction between two applications (the SW and HW simulators), like the OS support for co-simulation, as well as the remote interface between the board and the HW simulator.
The proposed co-simulation framework has been successfully applied to the Ultimodule *SCM2x0* platform for factory automation [1], a RISC system based on an user configurable FPGA system on chip (*SOC*) and hosting a RTOS [2] to allow real time operations, that delivers optimized, industry-ready solutions. The framework provides designers with a low cost development tool to rapid prototyping model of new hardware, either to be driven from the *SCM2x0* or extending the module itself, thus actively interacting with final customers during the requirements collection and shortening the process of new projects implementation.
The paper is organized as follows. Section 2 review existing work on timed co-simulation; Section 3 describes the main features of the proposed co-simulation framework, whose technical issues are discussed in Sections 4 and 5. Finally, Section 6 reports some experimental results, and Section 7 is devoted to concluding remarks.

2. Previous Work

The literature on HW/SW co-simulation is quite vast, and covers many different involved aspects. Historically, HW/SW co-simulation has been mostly focused on *functional* simulation, in which timing information, when available, is obtained from a static timing analysis of the various components.
Simulation performance was the strongest limitation to the development of effective timing accurate co-simulation strategies. This is particularly true for heterogeneous approaches, where a significant overhead is imposed by the effort of keeping the synchronization between the two (the

Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE'05)
1530-1591/05 $ 20.00 IEEE

software and the hardware one) simulation engines ([3, 4, 5, 6]). Although *homogeneous* environments (in which a single simulation engine is used) may make this task more manageable, homogeneity is usually achieved by abstracting away the distinction between hardware and software, making the very notion of time quite imprecise ([7, 8]).
Only recently, some approaches have addressed this performance bottleneck by explicitly targeting the timing accuracy of co-simulation.
One class of approaches borrows ideas from the theory of distributed synchronization, using the similarity between timed co-simulation and distributed event-driven simulation algorithms. [9, 11, 12, 13]. All these approaches follow an *asynchronous* paradigm, in which each simulator manages its local time, and the local times evolve at different speeds. They differ, however, in how the overhead required by synchronization is managed; the solutions consider either the use of rollback of the simulation (when one simulator receives a past event from the other simulator – [9]), or a proper alignment between the local clocks and global clock, when rollback is not possible, allowing synchronization to occur only when events are exchanged ([11, 12, 13]).
Another class of solutions is based on the construction of a timing model for software, obtained by attaching timing annotations to the ISS (for instance, an execution time in cycles for each executed instruction). Timing synchronization between software and hardware is then achieved using the accumulated delays for the software, and the cycle information provided by a HDL simulator for the hardware [14, 15].
All the above approaches provide a reasonably effective solution to the timed co-simulation problem. However, they are explicitly targeted for the interaction of two (or more) co-simulation engines, and are not thus suitable for our virtual prototyping context, which requires the timing-accurate interfacing of a simulator for hardware and a software program running on an actual board, possibly hosting a RTOS. In other terms, we need synchronization between *actual* (i.e., not *simulated*) time (on the board side where SW is executed) and simulated time (on the host computer side where an HW model is simulated).
This requirement rules out many of the possibilities previously reviewed; options such as simulation rollback or the use of instruction-based timing models, for instance, are either infeasible or inadequate. In fact, the real-time execution of the software on the board is based on the synchronization given by the hardware timer which monotonically increases its value. Moreover, the board may include some hardware devices which synchronizes their work by exploiting the timer value, thus rollback cannot be implemented, since it would require the rollback of the behavior of such real hardware components. The proposed co-simulation methodology solves this problem by defining the concept of *virtual tick*.

3. Co-Simulation Methodology

The objective of this work is the timed HW/SW co-simulation of a system in which a portion of the hardware is described in some HDL and the software is running on a microprocessor-based board. Figure 1 shows the specific instance of the system considered in this work.

The board includes a CPU, memory, and various hardware devices; some devices are conventionally implemented in hardware (ASICs), while some others by means of reconfigurable hardware (a FPGA hosted on the board). The CPU runs application software that accesses hardware functionalities by using device drivers provided by a RTOS. A hardware timer produces the signal that increments the clock counter used by SW and HW functions to synchronize their execution, thus allowing real-time behavior.

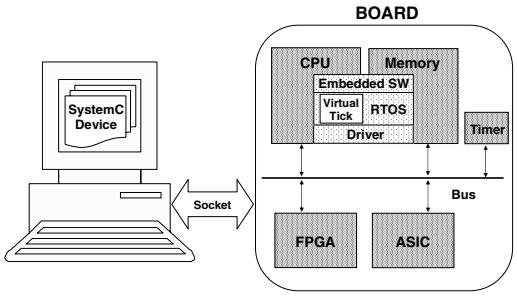

Figure 1. Co-Simulation Methodology.

Virtual HW prototyping is implemented by executing the hardware model on a host computer by means of simulation of a given HDL model. We selected SystemC as HDL, since it allows to model a hardware behavior at different abstraction levels. Moreover, a synthesis flow easily transforms HW models into HW devices.
Two are distinguishing features of the proposed co-simulation methodology:

1. *Communication between HW (the SystemC simulator) and the SW (the board) is realized by means of a* driver-based *scheme.* More specifically, the SW accesses to the HW device under design trough a device driver used to send and receive data to/from the SystemC model, viewed as any other device.

   Such scheme implies that communication between HW and SW occurs through the interrupt mechanism. However, due to the "remote" nature of the virtual HW device, the interaction between the board and the hardware simulator on the host PC is implemented by means of the network communication APIs of the OS. Similarly to previous schemes ([19, 20]) the proposed methodology uses the remote IPC to implement the interaction between SW and HW models; however, the use of a real board and the requirement of timed HW-SW co-simulation completely differentiates the methodology proposed in this work. In particular, our scheme can be viewed as a variant of the solution originally proposed in [20]; the details will be described in Section 5. Moreover, with respect to the paper [21] the proposed framework estabilishes a complete timed co-simulation between the HW model and all the SW running on the board, while [21] allows to synchronize the HW model with the application SW thread only. Performance estimation of the proposed methodology are thus more accurate.

2. *Timing synchronization is realized using the notion of* virtual tick. The latter term identifies the fact that the co-simulation is driven by the SystemC simulated time (a virtual time), rather than the real time of the timer on the board. This implies thus that (i) SystemC becomes the master of the co-simulation, and (ii) the real-time behavior of the HW/SW components of the board will actually depend on the SystemC virtual clock rather than the actual clock counter that is updated by the RTOS timer interrupt routine. In other terms, in hardware terms, it is the (SystemC) device that determines the advance of time.

   Given the different natures of the two times, the continuous-time evolution of the timer (on the board) becomes a discrete-time evolution based on the simulated time.

4. Timing Synchronization Protocol

The previous section has described the main features of the proposed timed co-simulation methodology. In this section, we concentrate on the timing issues of the co-simulation, and in particular on how to synchronize the hardware timer with the simulated time of the SystemC kernel.

4.1. Enforcing Timing Synchronization

The hardware timer hosted on the board generates periodic pulses (*HW ticks*); software operations on the board are synchronized with respect to a *SW tick*, which is derived by interrupting the system after a specified amount of HW ticks. When the interrupt occurs, a timer interrupt service routine is invoked, which increments the SW tick counter and takes care of the alarms, and eventually calls the scheduler to provide it with the time-slice use for scheduling decisions.

The key point of our approach is that *timing-accurate cosimulation is realized by altering the way the SW tick is updated*; rather than modifying it any time the HW timer generates an interrupt, the changes are driven by the SystemC clock.

The idea, summarized in Figure 2, is that the SystemC simulator sends, in correspondence of each SystemC clock cycle, a "message" to the board to force the update of the SW tick. The message actually appears to the board as a regular device-driven interrupt request. When the message has been sent, the SystemC simulator must wait for a response message before starting the next simulation cycle.

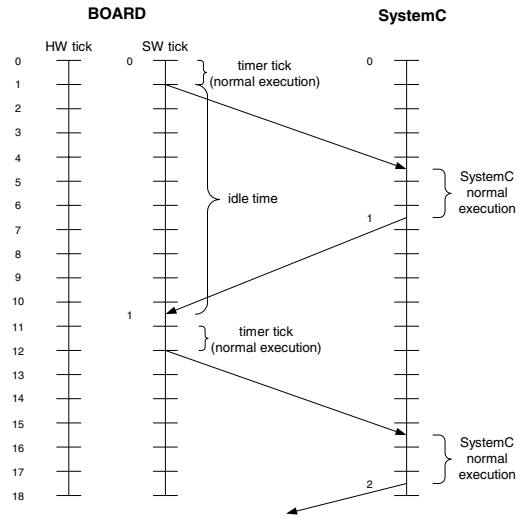

Figure 2. Co-Simulation timing view.

In order to implement this idea, however, it is essential to observe that the interval between two HW ticks (1 ms in the Figure 2) can be shorter than the time required to send a message (via socket) from the host running the SystemC simulator to the board.

Therefore, after one SW tick, the board must spend some time (denoted with *idle time* in the following) for waiting the arrival of the message from the SystemC simulator. During this idle time, the board must put all system and user threads into a sleep step, but those that are necessary to keep the socket connection between the Board and SystemC active.

4.2. Reducing IPC Overhead

In the previous description, the SW tick is incremented for each increment of the SystemC clock. This scheme is clearly highly inefficient, because it requires an excessive amount of synchronization. To reduce the communication overhead, we limit the number of required synchronization points. This can be achieved by having SystemC to send the synchronization message only after *n* clock cycles. This implies that the board (actually, the OS), upon arrival of this "multiple-tick" message, must run for *n* SW ticks.

We call *synchronization time* (T_{sync}), the interval (measured in SystemC clock cycles) between two synchronization events which are sent from SystemC to the board.

This is obviously possible only if there no exchange of data between the board and the SystemC simulator occurs during normal execution.

5. Implementation Details

The implementation of the proposed methodology requires three main tasks. First, the definition of the communication interface; second, the modification of the OS running on the board, in order to support the board, and, third, the modification of SystemC simulation kernel, in order to setup the proper synchronization commands. In such a way the synchronization (but also the exchange of data) between the two components is properly managed. This section details the three above issues.

5.1. Implementing the Communication

The generic architecture of the proposed methodology shown in Figure 1 shows a generic socket-based connection between the two actors involved in the communication. The latter is specifically implemented through three TCP/IP ports: A (`DATA_PORT`), used to exchange data, An interrupt port (`INT_PORT`), which carries the signal interrupt, and a clock port (`CLOCK_PORT`), which takes care of the exchange of the timing information that keep the two systems synchronized.

Whenever a synchronization event goes from the SystemC kernel to the board, the OS running on the board advances the application for T_{sync} cycles. At the same time the SystemC kernel advances its simulation for T_{sync} cycles and then it waits an answer from the board. Therefore, when a time packet is exchanged between the two actors, they are fully synchronized and they can further proceed. Also, whenever an interrupt is sent from the SystemC environment, the OS reads and writes on the `DATA_PORT` to exchange data with the simulated hardware.

5.2. SystemC Kernel Modifications

The synchronization between the board and SystemC is realized by modifying the SystemC kernel in such a way that it can establish and control the communication. The required modifications to the SystemC kernel essentially consist of:

- The addition of two new types of ports (`driver_in` and `driver_out`), that are devoted exclusively to the communication between a SystemC module and the OS running on the board. These classes are derived from the `sc_in` and `sc_out` SystemC classes, respectively.
- The addition of the special process `driver_process`. Similarly to a `sc_method`, a `driver_process` will be triggered when a new data is present on a `driver_in` port to which the process is sensitive.

- The modification of the event scheduling algorithm, in order to handle the presence of special ports and processes.
- The adding of an additional module to the design, triggered by the positive edge of the clock, which sends and receives the synchronization packets from the board.

The core of a standard SystemC simulation engine is enclosed in a function called `simulate`. Such a function is invoked when the simulation starts, and it runs the simulation loop, calling the proper routines to collect, handle and dispatch events.
In the modified version of the SystemC kernel, we replaced that function with another one, called `driver_simulate`. This new function opens the communication channels, opening the TCP/IP `DATA_PORT` and `INT_PORT` ports. Then it iterates on a modified simulation loop, which performs the following actions:

- Check for the presence of data on `DATA_PORT`. If there is one, the datum is read and the required actions are performed. These actions are:
  - Reading from the SystemC port;
  - Sending data to the board;

  in case of a read, and:
  - Reading further data from the channel.
  - Writing the SystemC ports.
  - Advancing the `driver_process`;

  in case of a write.
- A standard simulation cycle is accomplished. As in the conventional SystemC simulation step, events are handled and generated. New events are scheduled and dispatched.
- The interrupt signal is checked. If it is active, a packet is sent to the board via the `INT_PORT`, so the board will react by sending a request packet on the data channel.

5.3. OS Modifications

There are two main modifications required by the OS.
The first one is concerned with the *communication* between the board and the hardware. Since the simulated hardware is modeled as an external device, this amounts to write a new device driver to the OS. The driver is initialized at system boot, and it passively listens to the interrupt signal. As for any other physical device, activation of the interrupt signal notifies that the simulated device has some data to exchange, so the driver routines are in charge to read data from the `DATA_PORT`. Whenever data are read, they are transferred and elaborated by the proper function. Then, if needed, the results are sent back to the device, writing on the `DATA_PORT`.
The second main modification is concerned with the enforcement of the timing *synchronization*. To keep the software running on the board temporally aligned with the hardware simulated in the SystemC framework, the execution of the application must be driven by the same clock of the SystemC simulation.
The idea is to freeze the board until a SystemC clock pulse is received, then allow the execution for T_{sync} cycles and then block the board again. Since a timing packet from the SystemC side has to be received on a TCP/IP port, however, such approach is not viable. In order to receive and manage the clock pulse, in fact, the board cannot be completely idle, and some tasks on the board must be kept running.

Rather than completely stopping the entire board, however, we can identify what are the set of OS threads that are in charge to control the communication and that must thus keep always active. We called such threads *communication threads*. Once the latter have been identified, we split the behavior of the OS in two major states: `normal` and `idle`.

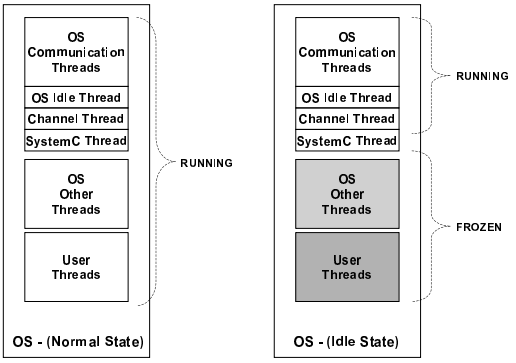

Figure 3. OS Normal and Idle States.

In the normal state the OS acts in a conventional way, scheduling user and system threads according to their priorities. In the `idle` state, conversely, only a limited number of tasks are allowed to run: the *communication threads*, the `idle_thread`, the `channel_thread` and the `systemc_thread` (Figure 3).
The `idle_thread` is a thread provided by almost all OSs, whose only purpose is to waste CPU cycles when no elaboration has to be performed.
The `channel_thread` is a thread that is spawned when the device driver is initialized, and listens on the `INT_PORT`, through which it performs the communication. The `channel_thread` cannot be halted when the OS is in the `idle` state, otherwise some events can be lost. Clearly, only the data exchange is performed by this thread, while the data management is left to other threads that run only when the OS is in the `running` state.
Last, the `systemc_thread` is the special thread that we add to the OS for synchronization purposes. It is in charge of listening to the `CLOCK_PORT` for an incoming clock pulse from the SystemC side; when such an event occurs, it reacts accordingly.
The typical sequence of operations of the OS works as follows. The `normal` state should last for a time at least equal to T_{synch} cycles. The timer interrupt, which is invoked on regular intervals, is used to handle the switch from the `normal` to the `idle` state. When it is time for such a state switch, the following steps are performed:

- A flag is set.
- The need for a re-scheduling of the tasks is signaled to the scheduler;
- The scheduler saves the context (in particular, the value of the timeslice) of the thread currently in execution;
- The scheduler activates one of the threads allowed in `idle` state.
- The current time of the board is sent back to SystemC, to signal that the OS is frozen.

The `idle` state continues until a packet is received on the `CLOCK_PORT` from the SystemC side. Upon occurrence of this event:

- The above mentioned flag is cleared;
- The scheduler is invoked;
- The scheduler resumes the thread that was suspended and restores its context (in particular, the value of its timeslice).

Figure 4 summarizes the events that cause the OS to switch between the two states.

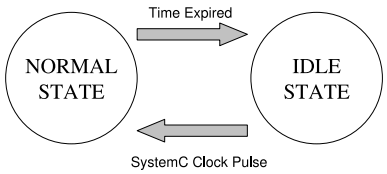

Figure 4. Switching from Normal to Idle and Viceversa.

6. Experimental Analysis

The co-simulation methodology presented in this paper has been used to verify the design of a small 4-port router described in SystemC, an extension of the Multicast Helix Packet Switch example distributed with SystemC 2.0.1.
This router receives data packets on its input ports and forwards them to the proper output port according to a routing table embedded into the router. Whenever a new packet arrives on one of the input ports, it is stored into an internal buffer. If the buffer is full, the packet is dropped. Each packet is then read from the buffer by the main process of the router, and checked for errors by a checksum algorithm. If the checksum is correct the destination address stored in the packet is used to find the right output port using the routing table; otherwise the packet is dropped. The packets consist of the following fields:

- *Source address*: the address of the producer.
- *Destination address*: the address of the consumer to which the packet must be sent;
- *Packet identifier*: an integer value used for debugging purposes only;
- *Data field*.
- *Checksum*: a 16 bit field used for error detection.

The overall HW/SW configuration consists of the following entities:

- *SystemC model of the router*;
- *SystemC model of the packet generator (producer)*, which is attached to an input port of the router, and generates packets with a random destination address.
- *SystemC model of the packet destination (consumer)*, which is attached to an output port of the router, and analyzes the integrity of the received packet.
- *C application computing the checksum*, executing on a SCM220 Ultimodule™ board running the eCos operating system.

In order to enable the proposed methodology, we have created a special device driver for the router and embedded it into eCos; the C source code calls the appropriate driver interface functions to communicate with the SystemC module. In addition, the modifications described in Section 5 have been added to eCos.

6.1. Co-Simulation Overhead

The first experiment is concerned with the evaluation of the overhead of synchronization. Figure 5 reports the overall time (in seconds) as a function of the number of exchanged packets N, for different values of T_{synch}.

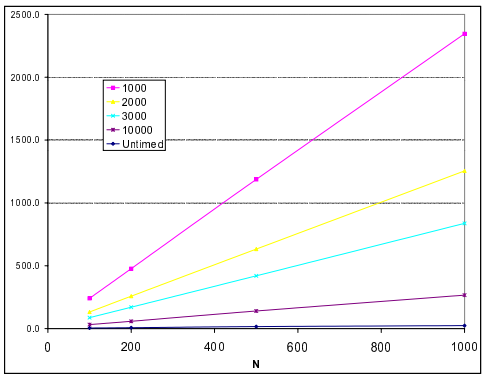

Figure 5. Co-Simulation Overhead.

The increase in the slope of the curves for decreasing values of T_{synch} in the plot shows that the overhead increases as the synchronization become tighter. The increase is due to (i) the increased cost of communication, and (ii) the overhead in the OS, which spends more time in switching between the *running* and the *idle* state.
The plot also shows an interesting feature of the proposed co-simulation scheme, namely, the fact that simulation time increases linearly with N, whatever the value of T_{sync}. This implies that the overhead is approximately constant, and does not depend on the number of exchanged packets. For instance, simulating the transmission of $N = 100$ packet takes 241 seconds for $T_{sync} = 1000$ and 32 seconds for $T_{sync} = 10000$, corresponding to a ratio of $241/32 \approx 8$, which also for any other value of N.
Figure 6 quantifies the overhead more precisely, by showing its direct dependency on T_{synch}; values are now expressed as the ratio between the actual simulation time (for a given T_{synch}) and the time spent by a simulation without synchronization (corresponding to $T_{synch} \rightarrow \infty$), assumed to be equal to 1.
The plots confirms the results of Figure 6, but emphasizes the fact that overhead decreases quite rapidly for larger values of T_{synch} (notice the log-scale on the Y-axis). For instance, imposing synchronization at each simulation cycle yields a simulation time which 1000x the time required for an untimed simulation; this overhead decreases to 100x if we synchronize once every 360 cycles.
The plot also shows that changing the amount of work done ($N = 100$ and $N = 1000$ exchanged packets) does not significantly change the rate at which the overhead decreases.

6.2. Co-Simulation Accuracy

The seconds experiment aims at assessing the simulation accuracy as a function of T_{sync}. The accuracy is expressed in terms of the percentage of packets that can be handled by the system. This number is 100% when the systems are very tightly coupled (a synchronization event for each simulated cycle), and it expected to progressively decrease as the synchronization becomes more loosely coupled. The plots of Figure 7 confirm this expected behavior. Notice however

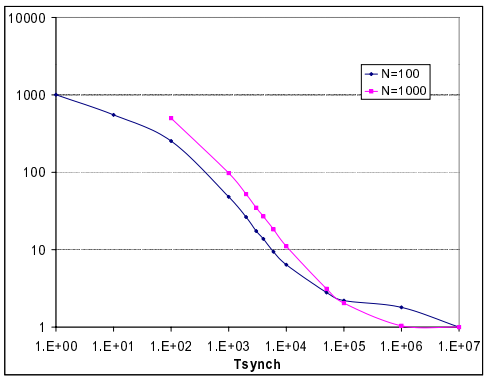

Figure 6. Co-Simulation Overhead vs. T_{sync}

that the 100% percentage of forwarded packets is maintained up to a value of T_{sync} around 5000, allowing thus a significant reduction of the simulation overhead.
The similarity of the two curves for $N = 100$ and $N = 1000$ shows that there is only a marginal degradation of the accuracy when increasing the number of packets sent. The small difference is due to the fact that dropped packets tend to increase when there is more work to be done.

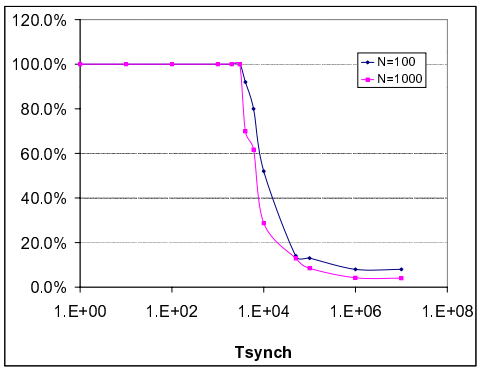

Figure 7. Simulation Accuracy vs. T_{sync}**.**

One final remarks is related to the choice of T_{synch}. In many cases, the value of T_{synch} is precisely defined by the timing constraints imposed by the device to be designed, and there is thus little margin to change it.
There may be however situations in which T_{synch} is allowed to vary over a given range of values. In these cases, the plots of Figure 6 and Figure 7 can be used together to define with an optimal value of T_{synch}. In fact, because of the opposite dependencies of the overhead and of the accuracy on T_{synch}, there is a value of T_{synch} which maximizes the product (accuracy x overhead). If the optimal value falls in the allowed range, the designer may then use it as the synchronization interval for the co-simulation.

7. Conclusions

We have proposed a new HW/SW co-simulation methodology which is suitable for virtual prototyping of new hardware devices to be added to an existing processor-based system.
Two are the main features of the proposed scheme. First, the time-accurate nature of the co-simulation, which allows to debug the device under design with the precision of the target hardware simulator. Second, thanks to the features of OS running on the board, the co-simulation even allows to meet real-time constraints, provided that these are relative to the same timing reference used by the board.
The proposed framework has been successfully applied to a Ultimodule *SCM2x0* board to design a device implementing the function of a packet router, and to carry out a design exploration before implementing the device onto the board.

References

[1] Ultimodule, Inc., `http://www.ultimodule.com`.

[2] RedHat eCos, `http://sources.redhat.com/eCos`.

[3] A. Ghosh et al., "A Hardware-Software Co-Simulator for Embedded System Design and Debugging," *ASPDAC'95*, pp. 155-164, January 2000.

[4] C. Liem, F. Nacabal, C. Valderrama, P. Paulin, A. Jerraya, "System-on-Chip Co-Simulation and Compilation," *IEEE Design and Test*, Vol. 14, No. 2, pp. 16-25, Apr.–Jun. 1997.

[5] C. Valderrama, F. Nacabal, P. Paulin, A. Jerraya, "Automatic VHDL-C Interface Generation for Distributed Co-Simulation: Application to Large Design Examples", *Design Automation for Embedded Systems*, Vol. 3, No. 2/3, pp. 199–217, March 1998.

[6] P. Coste, F. Hessel, Ph. Le Marrec, Z. Sugar, M. Romdhani, R. Suescun, N. Zergainoh, A. Jerraya, "Multilanguage Design of Heterogeneous Systems", *International Workshop on Hardware-Software Codesign*, pp. 54–58, May 1999.

[7] J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt, "Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems," *International Journal of Computer Simulation*, Vol. 4, pp. 155–182, April 1994.

[8] F. Balarin et al., *Hardware-Software Co-Design of Embedded Systems: The Polis Approach*, Kluwer Academic Press, 1997.

[9] S. Yoo, K. Choi, "Optimistic Timed HW-SW Cosimulation", *ASPHDL'97: Asia-Pacifi c Conference on Hardware Description Language*, Aug. 1997, pp. 39–42.

[10] R. Righter, J. C. Walrand, Distributed Simulation of Discrete Event Systems," *Proceedings of the IEEE*, Vol. 77, No. 1, Jan. 1995.

[11] W. Sung, S. Ha, "Optimized Timed Hardware Software Cosimulation Without Roll-Back," *DATE'98: Design Automation and Test in Europe*, Mar. 1998, pp. 945–946.

[12] D. Kim et al., "Virtual Synchronization for Fast Distributed Cosimulation of Dataflow Task Graphs," *ISSS'02: International Symposium on Systems Synthesis*, Oct. 2002, pp. 174–179.

[13] Y. Yi, D. Kim, S. Ha, "Virtual Synchronization Technique with OS Modeling for Fast and Time-Accurate Cosimulation," *CODES/ISSS'03: International Workshop on Hardware-Software Codesign*, pp. 1–6, Oct. 2003.

[14] S. Yoo, G. Nicolescu, L. Gauthier, A.A. Jerraya, "Fast Timed Cosimulation of HW/SW Implementation of Embedded Multiprocessor SoC Communication," *HLDVT'01: IEEE International Workshop on High Level Design Validation and Test*, Oct. 2001, pp. 79–82.

[15] I. Bacivarov, S. Yoo, A. A. Jerraya, "Timed HW-SW Cosimulation Using Native Execution of OS and Applications SW," *HLDVT'02: IEEE International Workshop on High Level Design Validation and Test*, Oct. 2002, pp. 51–56.

[16] J. Liu, M. Lajolo, A. Sangiovanni-Vincentelli, "Software Timing Analysis Using HW/SW Co-Simulation and Instruction Set Simulator," *CODES'98: International Workshop on Hardware-Software Codesign*, pp. 65–69, Mar. 1998.

[17] L. Semeria, A. Ghosh, "Methodology for Hardware/Software Co-verifi cation in C/C++", *ASPDAC'00*, pp. 405–408, Jan. 2000.

[18] K. Lahiri, A. Raghunathan, G. Lakshminarayana, S. Dey, "Communication Architecture Tuners: a Methodology for the Design of High-Performance Communication Architectures for System-on-Chips," *DAC-37*, pp. 513-518, Jun. 2000.

[19] L. Benini, D. Bertozzi, D. Bruni, N. Drago, F. Fummi, M. Poncino, "SystemC Co-simulation and Emulation of Multi-Processor SoC Designs," *IEEE Computer*, Vol. 36, No. 4, Apr. 2003, pp. 53–59.

[20] F. Fummi, S. Martini, G. Perbellini, M. Poncino, "Native ISS-SystemC Integration for the Co-Simulation of Multi-Processor SoC," *DATE'04: Design Automation and Test in Europe*, Feb. 2004, pp. 564-569.

[21] L. Formaggio, F. Fummi, G. Pravadelli, "A Timing-Accurate HW/SW Co-Simulation of an ISS with SystemC," *CODES+ISSS 2004: International Conference on Hardware Software Codesign and System Synthesis*, Sep. 2004, pp. 152-157.

[22] Synopsys, Inc., "SystemC, Version 2.0", *http://www.systemc.org*.