A Timing-Accurate HW/SW Co-simulation of an ISS with SystemC

Luca Formaggio Franco Fummi Graziano Pravadelli
Dipartimento di Informatica – Università di Verona
Strada le Grazie 15, 37134 Verona, Italy
luca.formaggio@students.univr.it franco.fummi@univr.it pravadelli@sci.univr.it

ABSTRACT

The paper presents a system level co-simulation methodology for modeling, validating, and analyzing the performance of embedded systems. The proposed solution relies on the integration between an instruction set simulator (ISS) and the SystemC simulation kernel. In this way, the ISS is used to abstract the model of the real programmable device where the SW should run, while SystemC is used to model HW components that interact with the SW. A correct validation of such an architecture is infeasible without taking care of timing information. Thus, the paper proposes an effective timing synchronization mechanism, which uses timing information of an ISS (or a board) to synchronize the SystemC simulation.

Categories and Subject Descriptors: B.8.2 [Performance and Reliability]: Performance Analysis and Design Aids

General Terms: Design, Performance.

Keywords: System level modeling, Co-simulation.

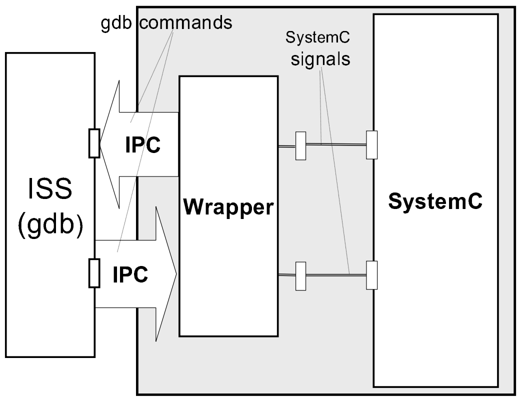


Figure 1: SystemC wrapper and ISS.

1. INTRODUCTION

The integration of HW and SW components is one of the most challenging tasks in system-level design of embedded systems. An accurate co-simulation technique is mandatory to validate the behavior and evaluate the performance of the whole system at the early stages of the design flow. For this reason, several co-simulation platforms [1, 2, 3, 4, 5, 6, 7, 8] have been developed in the past years. In spite of the variety of architectural targets, performance efficiency and description languages, we can classified these different solutions in two main categories: *homogeneous* co-simulation environment and *heterogeneous* co-simulation environment.

Homogeneous environments use a single engine for the simulation of both HW and SW components. The Ptolemy [1] and Polis [2] environments are pioneering works in that direction. In these approaches, homogeneity is achieved by abstracting away the distinction between hardware and software parts that are described as functional blocks. Homogeneous environments simplify the design modeling and they provide good simulation performance. However, they are suitable only in a very initial phase of the design, prior to HW/SW partitioning. On the contrary, heterogeneous environments ensure a more accurate tuning between HW and SW components. Most of these frameworks essentially address the same problem: how to efficiently link event-driven hardware simulators and cycle-based instruction set simulators. Earlier HW/SW co-simulation frameworks [5, 6, 7] are mainly focused on multi-language system descriptions, i.e., HDL for hardware description, and programming language for software. All these approaches are quite similar in that, as heterogeneous co-simulation solutions, their main effort aims to solve the issue of controlling and synchronizing two (or more) simulation engines. This heterogeneous style is sub-optimal in terms of simulation performance and easiness of integration, but it was the only possible choice when VHDL or Verilog simulation was the highest possible level of abstraction for simulating hardware. Some commercial tools, such as Mentor Graphics Seamless [4] and Synopsys Eaglei [3], also provide heterogeneous co-simulation capabilities. However, they allow HW/SW co-simulation at bus level, where each bus transaction involves all signals necessary to accomplish the bus function, thus degrading the co-simulation performance.

The advent of design flows based on SystemC allowed the definition of efficient *semi-homogeneous* approaches [9, 10, 11, 12, 13], where the bus is abstracted to obtain a more efficient co-simulation. They are homogeneous from the language point of view, since both HW and SW are described by using C++. This definitely simplifies the implementation of the initial model as well as the subsequent HW/SW partitioning. However, these approaches are heterogeneous from the simulation point of view, since HW and SW can be executed by using different simulators: the SystemC simulation kernel for the HW components and an ISS for the SW programs. In this way, a more accurate performance estimation can be performed, since the heterogeneous model reflects the final embedded system. All these environments are based on two basic concepts (Figure 1):

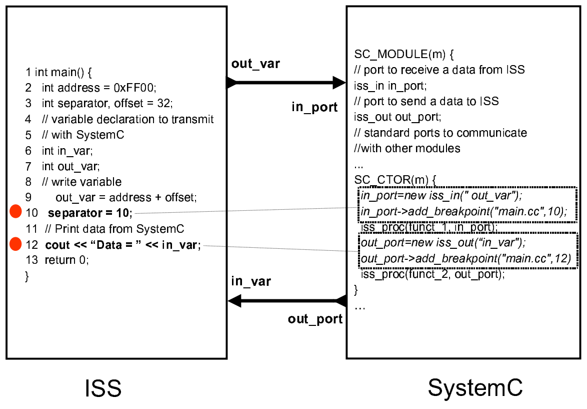
- *Interprocess communication (IPC)*. It is used to realize the communication between the Instruction Set Simulator (ISS), where the SW part runs, and the SystemC simulator, that model the HW part. The simulators run as distinct processes on a host system.
- *Bus wrapper*. It ensures synchronization between the system simulation and the ISS, and it translates the information coming from the ISS into cycle-accurate bus transactions.

Most of these approaches [9, 10, 11] define a *custom interface* between the bus wrappers and the ISS. This makes harder the integration of new processor cores within the co-simulation framework, because the ISS needs to be modified to support the IPC primitives defined by the co-simulation system. This issue is addressed in [8], where a standardized interface between bus wrapper and ISS is proposed. It is based on the remote debugging primitives of GDB [14]. In this way, any ISS that can communicate with GDB (that is, basically any) can also become part of a system-level co-simulation environment. The approach of [8] still suffers from some performance bottlenecks, since the ISS and the SystemC simulators evolve in lock-step, because synchronization is driven by the host operating system via IPC.

In [12] the authors solve some limitations of previous works by proposing two alternatives co-simulation methodologies that allow a SystemC description of hardware and an ISS to efficiently co-execute. The two proposed solutions differ with respect to the simulation kernel (SystemC or ISS) that drives the co-simulation. However, in both cases the interaction with SystemC simulation sessions is implemented at the kernel level, thus making it transparent to the SystemC code written by the user.

The main drawback of previous ISS-based co-simulation methodologies is represented by the lack of timing synchronization. This heavily reflects on a definitely not accurate design performance evaluation. On the contrary, timing information is considered in [15, 16]. These works propose a timing HW/SW co-simulation between native execution of operative system (OS) and application SW, and HW simulation. SW execution delay is annotated into the code of OS and application SW to allow HW synchronization. However, these solutions have two main drawbacks which limit an easy applicability: they are closely related to the specific implementation of the native OS, and the delay annotations depend on the processor where the SW runs (if the core model changes, delay annotations must be rewritten).

In this paper, starting from the GDB-Kernel approach presented in [12], we take care of timing information, thus providing a timing-accurate co-simulation mechanism. The proposed approach can be adopted to co-simulate SystemC modules describing HW components with a SW program



```
1 int main() {
2   int address = 0xFF00;
3   int separator, offset = 32;
4   // variable declaration to transmit
5   // with SystemC
6   int in_var;
7   int out_var;
8   // write variable
9      out_var = address + offset;
10  separator = 10;
11  // Print data from SystemC
12  cout << "Data = " << in_var;
13  return 0;
}
```

```
SC_MODULE(m) {
// port to receive a data from ISS
iss_in in_port;
// port to send a data to ISS
iss_out out_port;
// standard ports to communicate
//with other modules
...
SC_CTOR(m) {
 in_port=new iss_in(" out_var");
 in_port->add_breakpoint("main.cc",10);
  iss_proc(funct_1, in_port);
 out_port=new iss_out("in_var");
 out_port->add_breakpoint("main.cc",12)
  iss_proc(funct_2, out_port);
}
...
```

Figure 2: GDB-Kernel synchronization example.

running on an ISS or on a real board. To achieve this goal, the main solved problems have been:

- Definition of a simplified communication mechanism between ISS/board and SystemC which exploits SystemC-native `sc_port`.
- Definition of a communication protocol for exchanging timing information between the ISS/board and SystemC.
- Definition of a mechanism to keep timing synchronization, which has required the modification of the SystemC kernel.

The paper is organized as follows. Section 2 summarizes the GDB-Kernel approach upon which this work is built. Section 3 explains how a timing-accurate co-simulation between ISS/board and SystemC is achieved. Experimental results are shown in Section 4. They highlight the effectiveness of the methodology. Finally, concluding remarks are reported in Section 5.

2. GDB-KERNEL CO-SIMULATION

The GDB-kernel co-simulation approach assumes an architectural template consisting of several processors interacting with hardware blocks, and communicating between them through a common bus. Besides modeling hardware, the SystemC simulation kernel serves as a master that drives the overall co-simulation. It is based on the use of the GDB remote debugging interface (RDI) between the ISS and the wrapper used to connect ISS and the SystemC simulator. The wrapper can be seen as an extension of SystemC that makes HW/SW communication more efficient. Figure 1 summarizes these features.

The synchronization between ISS and SystemC is realized by modifying the SystemC kernel in such a way that it can establish and control the communication by using GDB commands. The required modifications to the SystemC kernel consist essentially of: (a) the addition of two type of ports `iss_in` and `iss_out`, that are devoted exclusively to the communication between a SystemC module and an ISS; (b) the addition of a special process called `iss_process` (similarly to a `sc_method`, an `iss_process` will start execution when a new data is ready on a `iss_in` port to which the process is sensitive); (c) the modification of the event scheduling algorithm to handle the presence of special ports and processes.

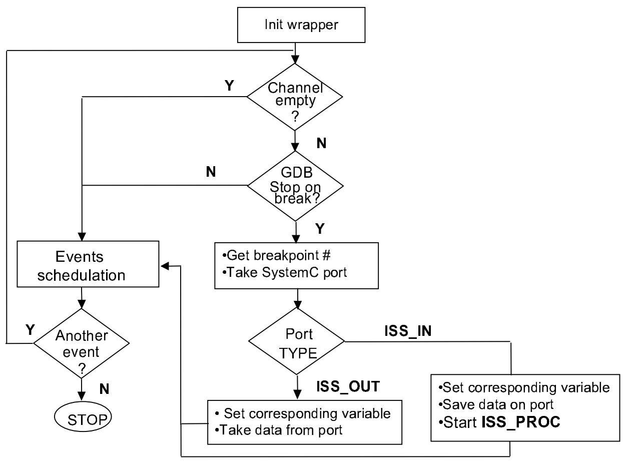


Figure 3: Modified scheduling algorithm.

From the ISS side, the interface between the SW program and SystemC is realized through ordinary variables, and it does not require any special modification. To set up the ISS/SystemC co-simulation, the programmer have to set breakpoints on the variables of the SW program that are considered as a channel to read or write data from the device described by the SystemC model. Then, the SystemC description has to be modified by defining `iss_in` and `iss_out` ports and associating them to the breakpoints previously set. Data exchange happens only when the SW is suspended on such breakpoints. Figure 2 shows a simple example of how variables and ports are matched to setup communication. Breakpoints at lines 10 and 12 allow, respectively, to connect the *separator* variable to the SystemC `in_port` and the SystemC `out_port` to the *in_var* variable. The control of the simulation is handled by the SystemC kernel. Figure 3 shows a high-level flowchart of the modifications of the SystemC scheduler. The algorithm, at the beginning of a simulation cycle, verifies if the GDB is stopped at a breakpoint by checking the content of the data structure of the IPC mechanism used to connect the ISS and the wrapper (i.e., a pipe). If not, the kernel does the normal handling of the events in the scheduler queue. Otherwise, the kernel checks at which breakpoint the GDB is stopped. If the breakpoint is associated to an `iss_in` port, a method of the wrapper class is used to get the new value of the variable from the ISS, then the value is stored into the corresponding `iss_in` port and the `iss_process`, sensitive to that port, is started. If the breakpoint is associated to an `iss_out` port, the value stored in this port is copied to the variable by using another method of the wrapper.

3. TIMING-ACCURATE CO-SIMULATION

Heterogeneous approaches require an ad-hoc synchronization mechanism to achieve a timing-accurate HW/SW co-simulation. In fact, differently form homogeneous solutions, the SW and the HW simulators do not share the same time schedule, because they are executed by distinct processes. In this section we show how the GDB-kernel co-simulation technique, described in Section 2, can be modified to implement a timing-accurate co-simulation framework, which is completely transparent to the designer.

3.1 Timing Information

To implement a timing-accurate synchronization mechanism, we modified the communication protocol between the ISS and the SystemC simulation kernel to exchange not only data, but also messages containing timing information. In particular, the SystemC simulation kernel, which drives the overall co-simulation, gets timing information from the ISS by means of GDB commands at each communication point (breakpoints in the SW code). Thus, the synchronization between the two simulators happens only when data exchange is needed; otherwise they run independently from each other. In this way, the synchronization overhead is reduced to the minimum.

Some ISSs, e.g., PSIM, the standard PowerPC emulator [17] integrated in the eCos environment [18], can be questioned about their execution time. PSIM provides the number of clock cycles elapsed since the SW program, that it is executing, has started. This number is a good estimation of the real elapsed time. Note that the co-simulation mechanism proposed in this paper can be adopted even if the SW is executed on a real programmable device mounted on a board, rather than on the ISS. In this case, timing information is extracted from the board, which definitely allows a correct computation of the SW execution time.

In real systems, HW and SW parts exchange data through a bus. Thus, to improve the co-simulation accuracy, we have defined a function that returns the number of clock cycles needed for data exchange, based on the bus type, the data size and the operation type (read or write). The return value of this function is added to the number of clock cycles provided by the ISS. In this way, the SystemC simulator can compute the correct execution time of the ISS/board by taking care of time needed for bus transfer too.

3.2 Timing Synchronization Issues

One of the following four cases (Figure 4) may happen when data are exchanged between the SystemC HW model and the SW running on the ISS:

Case 1. SystemC execution time is lower than ISS/board time and SystemC sends data to ISS/board.

Case 2. SystemC execution time is lower than ISS/board time and SystemC reads data from ISS/board.

Case 3. SystemC execution time is higher than ISS/board time and SystemC sends data to ISS/board.

Case 4. SystemC execution time is higher than ISS/board time and SystemC reads data from ISS/board.

According to the previous time inconsistencies, the SystemC kernel restores a correct timing synchronization by putting off SystemC events or forcing the ISS/board to waste time. This is implemented as follows:

Case 1. The SystemC event corresponding to data sending, and all events depending on it, are postponed to the ISS/board time. In the meanwhile the ISS/board is blocked on the breakpoint.

Case 2. The SystemC event corresponding to data reading is postponed to the ISS/board time. In the meanwhile the ISS/board is blocked on the breakpoint.

Case 3. The ISS/board must waste time until it equals the SystemC execution time, then SystemC can send data.

Case 4. The ISS/board must waste time until it equals the SystemC execution time, then SystemC can read data.

Section 3.3 describes how these operations are performed by modifying the SystemC simulation kernel.

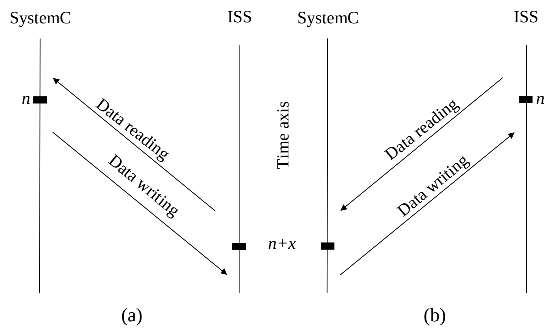


Figure 4: (a) Cases 1 and 2: SystemC execution time is lower than ISS time when data are exchanged. (b) Cases 3 and 4: SystemC execution time is higher than ISS time when data are exchanged.

3.3 SystemC Kernel Modifications

The implementation of the re-synchronization actions explained in the previous section demanded to modify the SystemC kernel described in Section 2. The required modifications consist essentially of the followings tasks:

- The definition of two types of ports, `iss_sc_in` and `iss_sc_out`, that replace the `iss_in` and `iss_out` ports of the original implementation. The `iss_sc_in` and `iss_sc_out` ports derive directly from the SystemC `sc_inout` port class. Thus, SystemC traditional processes (`SC_METHOD, SC_THREAD, ...`) can be declared sensitive to `iss_sc_in` and `iss_sc_out` ports without requiring ad hoc modifications to the SystemC events scheduling. The ISS-SystemC communication is simplified by using these new ports. On the contrary, the original implementation of the GDB-Kernel mechanism requires the definition of ad hoc processes (`ISS_PROC`) to use `iss_in` and `iss_out` ports, and a heavy modification of the kernel to manage `ISS_PROC` activation.
- The modification of the event scheduling algorithm, to implement the synchronization mechanism described in Section 3.2. It is worth to note, that the changes introduced to the SystemC kernel (see Figure 5) are not intrusive. They do not change the normal behavior of the kernel when events occur on signals not involved in the synchronization mechanism, thus not degrading the standard simulation performance.

Figure 5 shows the changes (dark shapes) introduced to the SystemC kernel with respect to Figure 3. When SW running on the ISS/board stops on a breakpoint, the SystemC kernel picks out the `iss_sc_port` necessary to exchange data with ISS, and before data exchanging, it restores the synchronization, according to the four cases described in Section 3.1.

Case 1 and case 2. SystemC needs to waste time before executing event `e` corresponding to data exchange. ISS/board is stopped on the breakpoint, while SystemC continues the normal events schedule. Event `e`, and all events depending on it, are postponed (Figure 5: *spend time* diamond) until SystemC time equals ISS/board time.

Case 3 and case 4. ISS/board needs to waste time before SystemC executes event `e` corresponding to data exchange

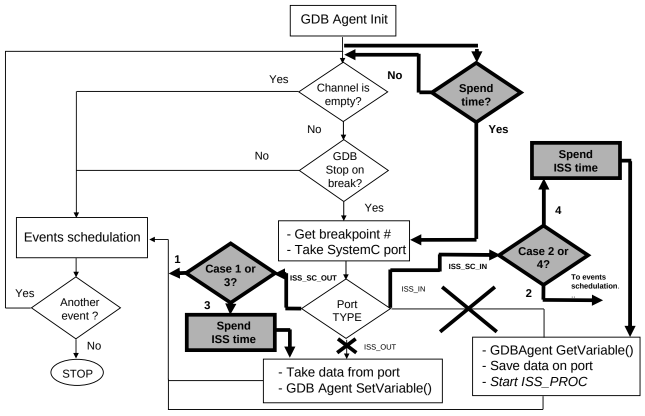


Figure 5: Modified scheduling algorithm for timing-accurate co-simulation.

```
1 void ConsumeTime(int clocks, int setup, int step)
2 {
3   int iterations_num;
4   iterations_num = (clocks - setup) / step;
5   for (; iterations_num >= 0; iterations_num--)
6   {
7     /*empty body*/
8   }
9 }
```

Figure 6: *ConsumeTime()* function.

(Figure 5: *spend ISS time* box). We have defined an ad hoc function to waste ISS/board time (Figure 6). The *ConsumeTime()* function is added to the file containing the SW program, and its execution is forced, by a GDB command, when the ISS/board is stopped on the breakpoint. The function body consists of an "empty" loop that is executed a number of times depending on the difference between the ISS/board and the SystemC time. Unfortunately the *ConsumeTime()* function cannot be cycle-accurate because the execution of each loop iteration takes more than one clock cycle. The *step* parameter specifies how many clock cycles are wasted by the ISS/board for each iteration (e.g., 8 cycles for PSIM). Moreover, the *setup* parameter specifies how many clock cycles are wasted by the function call, the instruction 4 and the function return. It represents the minimum number of clock cycles that can be wasted (e.g., 35 for PSIM). However, the little inaccuracy of the *ConsumeTime()* function does not heavily reflect on the co-simulation accuracy and performance. In any case, a perfect timing synchronization is achieved, since even if the ISS/board time overcomes the SystemC time, the latter is adjusted by following cases 1 and 2.

4. EXPERIMENTAL RESULTS

The accuracy of the proposed co-simulation approach has been proved by modeling an extended version of the *Mul-*

		Synchronization cases				
Simulated time		all	2,3,4	3,4	4	no sync
1 s.	forwarded packets	871	2383	2358	2315	2491
	get_ISS_time calls	6973	19070	18873	18523	0
	GDB commands	26845	60521	59798	56752	39867
	simulation time(sec)	28	70	69	68	19
10 s.	forwarded packets	16909	24045	25516	25156	25694
	get_ISS_time calls	135287	192389	204159	201284	0
	GDB commands	521137	672580	709591	640680	411161
	simulation time(sec)	523	689	750	744	197
100 s.	forwarded packets	169812	232897	236875	235403	249723
	get_ISS_time calls	1358680	1863480	1895290	1883530	0
	GDB commands	5226200	6644050	6761430	6046650	3996082
	simulation time (sec)	5721	7072	7205	7096	1928

Table 1: Co-simulation of the router for 1, 10 and 100 seconds of simulated time.

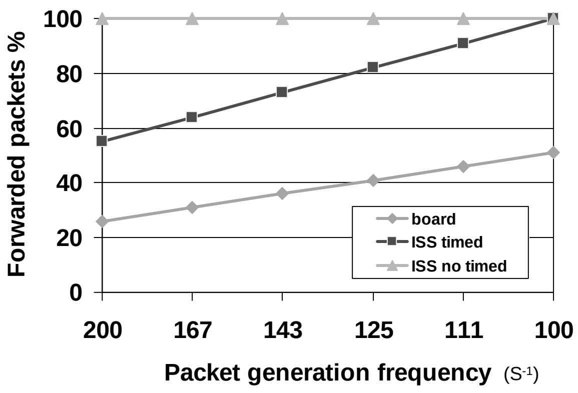


Figure 7: Percentage of forwarded packets.

ticast Helix Packet Switch[1] example distributed with SystemC. The router is modeled in SystemC. All packets coming into the router are buffered into a FIFO queue, then the main process takes the first packet in the queue and reads its destination address. Before sending the packet, according to the routing table, a checksum algorithm is computed on the packet to detect possible errors. The checksum is performed by a SW program running on the ISS, as commonly done in embedded routers. Breakpoints are defined in the source code of the checksum in correspondence to variables connected to `iss_sc_in` and `iss_sc_out` ports of the SystemC description. The environment is modeled in SystemC: the producers are modules that generate packets with a random destination address; the consumers are modules that analyze the integrity of the received packets.

Table 1 reports the co-simulation results for the router considering 1, 10 and 100 seconds of simulated time. Column *all* is related to the timing-accurate methodology proposed in this paper where all 4 synchronization cases are managed. The following columns report the results for co-simulations where the timing synchronization is provided only for the cases indicated in the column label. Moreover, the last column shows the result for the untimed approach. The rows of the table report the number of forwarded packets, the number of calls to the `get_ISS_time` function (which provides the execution time of the ISS), the number of GDB executed commands, and the simulation time. It is worth to note how the full timing-accurate approach does not dramatically affect the performance of the co-simulation. On average [2], the simulation time of the synchronized co-simulation is only 2.8 times higher with respect to the untimed co-simulation.

Despite of a little performance degradation, the timing-accurate mechanism gives a more precise performance estimation with respect to the case without synchronization. To evaluate the quality of the proposed co-simulation approach, we have analyzed the performance estimation achieved by running the checksum program on a real board, connected to the SystemC description of the router. The timing information extracted from the board allows a precise synchronization with the SystemC simulation kernel. Figure 7 compares the percentage of forwarded packets, with respect to the packet generation frequency, by applying three different co-simulation approaches: the ISS-SystemC untimed co-simulation mechanism, the timing-accurate ISS-SystemC approach proposed in this paper, and the same timing-accurate approach where the ISS has been replaced by a board with a real microprocessor. It is evident that the trend of the timing-accurate ISS/SystemC co-simulation is very similar to the board/SystemC co-simulation trend. By using an opportune scale factor, the error committed by considering the ISS instead of the real board is quite small. On the contrary, the number of forwarded packets by the untimed co-simulation is independent from the packet generation frequency. In such a case, a valuable performance estimation is infeasible.

The characteristics of the router does not allow us to analyze the co-simulation approach from the HW/SW partitioning point of view. Thus, we applied the methodology to another example: the *TPWire* network protocol [19]. The benchmark consists of a SystemC description modeling a TPWire chain with 126 slave nodes, and one master that transmits packets over the chain. The computation of some operations, related to packet data filling, can be extracted from the SystemC description and implemented in a C program executed by the ISS. Table 2 shows the experimental results of the co-simulation, when only one of these operations is implemented in SW.

Then, we evaluated a different HW/SW partitioning schema, where three operations of the protocol have been implemented in SW. Figure 8 shows the analysis performed by observing the variation of the transmission rate with re-

[1]In the following we call it *router*.

[2]For 10 and 100 seconds of simulated time.

	Synchronization cases				
	all	2,3,4	3,4	4	no sync
get_ISS_time calls	10551	11405	11145	11704	0
GDB commands	41969	39637	39221	35191	20661
simulation time(sec)	40	40	39	41	10
byte/sec	25	25	26	24	100

Table 2: Co-simulation of the TPWire protocol.

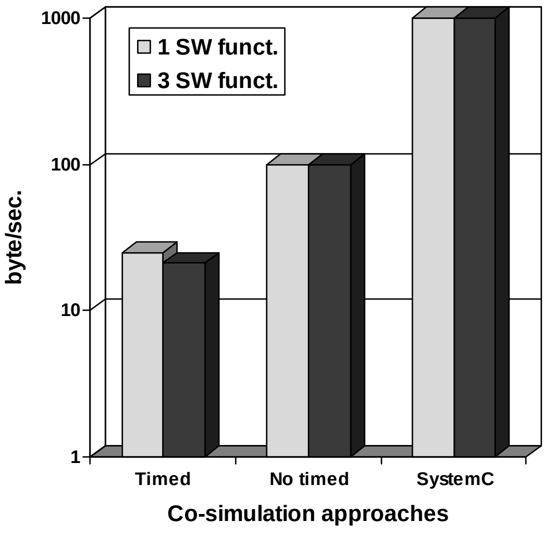


Figure 8: Transmission rate of the TPWire protocol.

spect to the partitioning schemas. It is evident that, the untimed co-simulation and the full SystemC simulation are not suited for HW/SW partitioning evaluation. In fact, they provide the same result independently from the number of SW functionalities. On the contrary, the proposed solution highlights that increasing the number of SW functionalities decreases the transmission rate.

5. CONCLUDING REMARKS

The paper described a timing-accurate co-simulation approach for modeling embedded systems, where the HW part is modeled by using SystemC, and the SW part is a program running on an ISS or on a real microprocessor. The synchronization mechanism is totally encapsuled into the SystemC kernel and completely transparent to the designer. Experimental results have highlighted that the proposed timing synchronization does not sensibly degrade the co-simulation time, while performance estimation of the SystemC/ISS becomes comparable with SystemC/board architecture. Finally, we have showed that the timing-accurate co-simulation allows a more precise design exploration for HW/SW partitioning.

6. REFERENCES

[1] J. Buck, S. Ha, E. Lee, and D. Messerschmitt. *Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems*. International Journal in Computer Simulation, vol. 4(2):pp. 155–182, 1994.

[2] F. Balarin, M. Chiodo, P.Giusto, H.Hsieh, A.Jurecska, L.Lavagno, C.Passerone, A.Sangiovanni-Vincentelli, E.Sentovich, K.Suzuki, and B.Tabbara. *Hardware-Software Co-Design of Embedded Systems: The Polis Approach*. Kluwer Academic Press, 1997.

[3] Synopsys Inc. *Eaglei*. Http://www.synopsys.com/products.

[4] Mentor Graphics Inc. *Seamless CVE*. Http://www.mentor.com/seamless.

[5] C. Liem, F. Nacabal, C. Valderrama, P. Paulin, and A. Jerraya. *System-on-Chip Co-simulation and Compilation*. IEEE Design and Test of Computers, vol. 14(2):pp. 16–25, 1997.

[6] C. Valderrama, F. Nacabal, P. Paulin, and A. Jerraya. *Automatic VHDL-C Interface Generation for Distributed Co-Simulation: Application to Large Design Examples*. Design Automation for Embedded Systems, vol. 3(2/3):pp. 199–217, 1998.

[7] P. Coste, F. Hessel, P. L. Marrec, Z. Sugar, M. Romdhani, R. Suescun, N. Zergainoh, and A. Jerraya. *Multilanguage Design of Heterogeneous Systems*. In *Proc. of IEEE International Workshop on Hardware-Software Codesign*, pp. 54–58. 1999.

[8] L. Benini, D. Bertozzi, D. Bruni, N. Drago, F. Fummi, and M. Poncino, *SystemC Co-simulation and Emulation of Multi-Processor SoC Designs*. IEEE Computer, vol. 36(4):pp. 53–59. 2003.

[9] J. Liu, M. Lajolo, and A. Sangiovanni-Vincentelli. *Software Timing Analysis Using HW/SW Co-Simulation and Instruction Set Simulator*. In *Proc. of IEEE International Workshop on Hardware/Software Co-design*, pp. 65–69. 1998.

[10] L. Semeria and A. Ghosh. *Methodology for Hardware/Software Co-verification in C/C++*. In *Proc. of IEEE Asian and South Pacific Design Automation Conference*, pp. 405–408. 2000.

[11] K. Lahiri, A. Raghunathan, G. Lakshminarayana, and S. Dey. *Communication Architecture Tuners: a Methodology for the Design of High-Performance Communication Architectures for System-on-Chips*. In *Proc. of ACM/IEEE Design Automation Conference*, pp. 513–518. 2000.

[12] F. Fummi, S. Martini, G. Perbellini and M. Poncino *Native ISS-SystemC Integration for the Co-simulation of Multi-Processors SoC*. In *Proc. of IEEE Design Automation and Test in Europe*, pp.564–569. 2004.

[13] I. Moussa, T. Grellier, and G. Nguyen. *Exploring SW Performance Using SoC Transaction-level Modelling*. In *Proc. of IEEE Design Automation and Test in Europe*, pp. 120–125. 2003.

[14] *GNU Project Web server*. Http://www.gnu.org/software/.

[15] S. Yoo, I. Bacivarov, A. Bouchhima, Y. Paviot, and A. Jerraya. *Building Fast and Accurate SW Simulation Models Based on Hardware Abstraction Layer and Simulation Environment Abstraction Layer*. In *Proc. of IEEE Design Automation and Test in Europe*, pp. 550–555. 2003.

[16] I. Bacivarov, S. Yoo, and A. Jerraya. *Timed HW-SW Cosimulation Using Native Execution of OS and Application SW*. In *Proc. of IEEE International High Level Design Validation and Test Workshop*, pp. 51–56. 2002.

[17] *PSIM User Guide and Reference Manual*. Http://sources.redhat.com/psim/manual/.

[18] *eCos Home Page*. Http://sources.redhat.com/ecos/.

[19] N. Drago, F. Fummi, M. Monguzzi, G. Perbellini, and M. Poncino. *Estimation of Bus Performance for a Tuplespace in an Embedded Architecture*. In *Proc. of IEEE Design Automation and Test in Europe*, pp. 188–193. 2003.