Addressing the Challenges of Synchronization/Communication and Debugging Support in Hardware/Software Cosimulation

Banit Agrawal Timothy Sherwood
Department of Computer Science
University of California, Santa Barbara
Email: {banit, sherwood}@cs.ucsb.edu

Chulho Shin Simon Yoon
ARM Inc., Irvine, CA
Email: {chulho.shin, simon.yoon}@arm.com

Abstract

With increasing adoption of Electronic System Level (ESL) tools, effective design and validation time has reduced to a considerable extent. Cosimulation is found to be a principal component of ESL tools to simulate the hardware designs and software models concurrently. It helps in providing an integrated system-on-chip design platform to get rid of most of the design errors in the early stage. To nail down these design errors early, a better debugging support of RTL memory on the software side is extremely useful. We present a just-in-time shadow memory technique that can allow debugging of RTL memory from a software perspective.

While cosimulation is fast compared to a complete hardware based simulation, the communication and synchronization overhead between the hardware and software simulators can be very significant. Since the two simulators have to communicate almost every cycle, a good communication platform is necessary to reduce this extra overhead. To evaluate this overhead, we implement and evaluate three communication primitives for a real system design with ARM926EJ-S processor and RTL memory. We find that a message-queue based communication backplane can alleviate the communication overhead to a considerable extent compared to other alternatives.

1. Introduction

Recently, software simulation tools (or ESL tools) are found to be very useful tools for the hardware designers to quickly explore a set of architectures with relaxed but acceptable timing accuracy [4]. The relaxed accuracy is acceptable because in the beginning stage of design, exploration speed is more important than accuracy to its traditional users. In hardware simulation, design is expressed in a hardware description language (HDL) and simulated in logic simulators (or HDL simulators). As this design is closer to hardware, it can provide more accurate results. But the overall design time and simulation speed makes embedded software development and validation impractical considering the ever-increasing scale of System-on-chip (SoC) designs. Hardware-software cosimulation is a hybrid approach where some IP models are expressed in a hardware description language and others are modeled in software. Cosimulation exploits benefits of both hardware simulation and software simulation to provide better flexibility and a fast simulation platform. It is found to be a viable solution to achieve fast and efficient architecture exploration. As HDL simulators require the models to be described in HDL, early-stage software models cannot be directly simulated with other hardware models. Similarly, software simulators require the models to be described in high-level languages like SystemC or C++, it is not straightforward to integrate with hardware models. Hence, cosimulation tools bridge the gap between hardware and software simulation and provide an architecture exploration platform for complete SoC design and in addition a verification platform where software models and hardware models can communicate. In ESL domain, developing an ESL model that accurately models the behavior of an existing hardware model takes significant effort. Though once the model is available, faster design exploration becomes possible. Transaction-level models (TLMs) can be integrated with already-available RTL IP models through cosimulation for validating a design, not requiring development efforts needed for building TLMs which in turn allows fast designer exploration.

Along with these advantages, there are some unique challenges associated with cosimulation such as how to provide better debugging support on the software side or how to achieve fast simulation speed by reducing the communication and synchronization overhead. Debugging on the software side can be extremely difficult if there is no debugging support for RTL memory components on the software side. Hence, a good RTL memory view capability is required. To this end, we present a just-in-time shadow memory technique (*patent pending*) that can provide better debugging capabilities for RTL memory on the software side. To address the communication overhead, we evaluate three communication primitives using a real example with ARM926EJ-S processor with RTL memory. We find that message queue based implementation can provide much faster cosimulation compared to other alternatives. Overall, our contributions in this paper are as follows:

- A just-in-time shadow memory technique is presented, which aids cosimulation by providing better debugging capabilities (memory view), fast cosimulation, disassembly support, and generating memory traces. (Section 3.1)
- To achieve fast simulation, we need to minimize the communication and synchronization overhead between hardware and software simulators. We implement and evaluate three communication primitives on a real system design to find the better communication backplane in different scenarios. (Sections 3.2 and 4)

In the next section, we discuss some of the related works. In Section 3, we present just-in-time shadow memory techniques for better debugging support and the implementation details of communication and synchronization in our cosimulation platform. In Section 4, we present the communication backplane evaluation to find the best available communication primitive. We conclude in Section 5.

2. Related Works

Cosimulation has been very attractive among the system designers and researchers since the last decade. The main challenges associated with hardware-software cosimulation are achieving faster simulation, better synchronization in heterogeneous cosimulation environments, visibility of internal state for debugging, getting better timing accuracy and availability/type of software models. The strength and weakness of various cosimulation techniques based on these challenges are compared in [8]. Most of these techniques are categorized based on the models used on the software and hardware side. Similarly, in [14] several key issues have been presented to combine the capabilities for software simulation and hardware simulation in a best possible manner.

Improving the speed of cosimulation with better timing accuracy is one of the most important challenges of cosimulation. Most of the previous works have focused on improving the speed of cosimulation [1,2,5,7,9,11,12,13], whereas only a few of them have looked into other different challenges [3,10]. Passerone et al. [7] proposed a technique to do fast cosimulation using constrained software synthesis that utilizes the run time estimation of a target processor. The estimation accuracy is limited by the caches, pipelined architectures, and communication cost. In [10], an integrated cosimulation environment is presented which allows the designers to model the entire system in one language C/C++. It does not interface with any HDL simulators. This technique limits the use of any third-party hardware core in the overall system design and verification process. A compiled simulation technique is presented in [13] that can generate bit-, cycle- and pin-accurate cosimulation engines which are much faster than the interpretive simulators. But any design changes require the recompilation of the system design. The techniques mentioned above are well suited in homogenous environments where only one integrated environment is used.

In heterogeneous environments, different simulators interact with each other that present a different set of problems compared to homogeneous environments. Kim et al. [1] presented an integrated and heterogeneous cosimulation environment that provides an automatic interface generation. Becker et al. [6] used distributed processes for cosimulation and tested their cosimulation environment by designing a network interface unit. The communication timing was less predictable in this approach due to communication between multiple distributed processes. Bishop et al. [2] present another heterogeneous cosimulation environment where time management and synchronization issues are addressed. In this scheme, as always, synchronization plays a big role in the overall cosimulation performance.

In [9], a trace-driven HW/SW cosimulation technique is proposed. In this paper, synchronization is addressed as major performance bottleneck to the system cosimulation. They alleviate the effect of the synchronization issues by using a virtual synchronization technique that makes use of the execution traces and the timing management of the execution traces. But in this scheme, the accuracy of cosimulation is dependent on the OS and channel model. Sung et al. [12] present a backplane approach for hardware-software cosimulation. This approach tries to minimize the communication of control data between the simulators.

A completely different approach is presented by Lee et al. [5] where architecture simulators inherit circuit modeling capabilities and react to circuit characteristics such as latency, energy on a per-cycle basis, and still provides a considerable throughput. This technique can be applied to existing cosimulation tools to get delay and energy estimation along with performance estimation. In a similar direction, a cosimulation based power estimation method is presented by Lajolo et al. [3]. Power estimation for different components of system-on-chip is done using concurrent and synchronized execution of multiple power estimators.

We use an ESL tool [15] that allows us to quickly evaluate the designs. It uses a SystemC core on the software side to provide fast cycle-based simulation and attempts to minimize the flow of information between the software simulator and RTL simulator. Since most of the time, the memory components are being simulated on the HDL side, it gets extremely slow and inflexible to debug the memory on the software side. To provide better debugging support of memory components, we present the just-in-shadow memory technique. Similarly, to address the issues of communication and synchronization overhead, we implement three communication primitives available on the Linux platform and in the next section we present its implementation details. Then, we use the ESL tool [15] to evaluate the overhead of a real system design with ARM926EJ-S processor.

3. Addressing Cosimulation Challenges

In this section, we address some of the key challenges in cosimulation by presenting the idea of just-in-time shadow memory that provides better debugging capabilities and also present how it is implemented in our synchronization and communication paradigm.

3.1 RTL Memory Debugging

One of the most common practices of cosimulation is that the core models run on an ESL simulator while memory subsystems (including the bus subsystems in most cases) are simulated on an RTL simulator. The reason being is that off-the-shelf ESL core models are more increasingly available in a mature state while the memory subsystem is likely to be the major target of a system design. In such cases, debugging gets difficult on the system simulation side because it doesn't know anything about the memory on the RTL side and it cannot provide any debug accesses to memory directly. Many times, protected IP cores are simulated on the RTL side and debug accesses to the RTL memory or RTL peripherals with memory-mapped registers gets extremely difficult. A just-in-time shadow memory[1] technique is presented, which provides a shadow memory on the ESL simulator side and it captures all debug writes to the RTL side and services all the debug reads. This technique is described in detail in the next subsection.

3.1.1 Just-in-time shadow memory

Just-in-time shadow memory technique[1] helps in providing a uniform memory view, debug read/write access to RTL memory, disassembly support. In this technique, an identical copy of the RTL memory is maintained on the ESL simulator. All the debug read requests from the ESL simulator are served from shadow memory instead of RTL memory. The debug writes from the ESL simulator has to be written to both shadow memory and RTL memory.

The key difficulties in maintaining the identical copy are debug writes from ESL simulator, normal writes from the ESL simulator, and writes on the RTL side. All of these

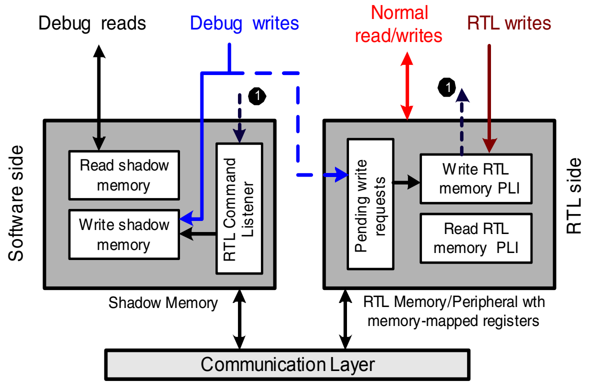


Figure 1: Just-in-time shadow memory block diagram. All the debug reads are read from shadow memory. Debug writes are written to shadow memory and pending writes are queued on RTL side. All the normal RTL writes are written back to shadow memory using a running thread on the ESL simulator side.

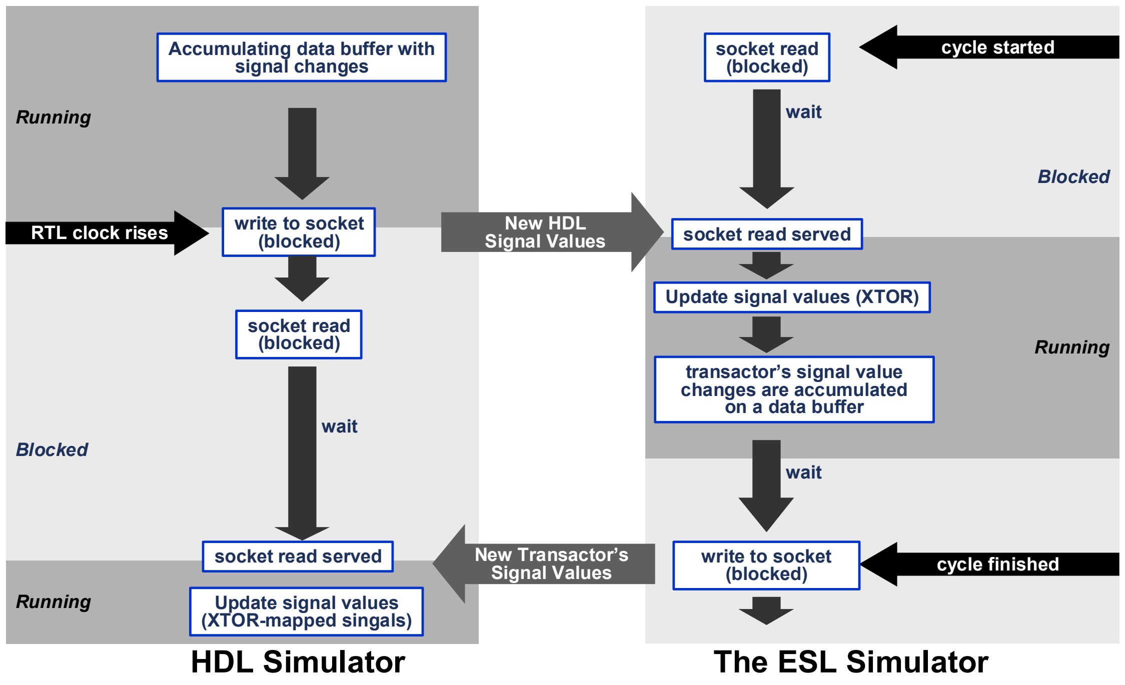
different types of write accesses are updated on both sides as shown in Figure 1.

The debug writes from ESL simulator are sent to the RTL side as memory write command using file-based sockets. Just before the starting of the next cycle, any pending write accesses are updated on the RTL side by the use of programming language interface (PLI) of the RTL simulator. In this case, it becomes necessary to know the signal name of memory object in RTL design to do PLI/FLI/VPI/VHPI accesses. We provide a cosimulation configuration interface where user can specify the signal name of the memory and type of object. The wrapper library on the RTL side looks for the memory signal names when it is loaded for the first time in the RTL simulator.

Normal writes through signals should also be updated on the shadow memory side. This requires little changes to the RTL memory code to do a PLI/FLI/VPI/VHPI access. This access is responsible to send a memory write command on the ESL simulator. A separate thread on ESL simulator side is listening for any write commands from the RTL side. On receiving the memory write command from the RTL side, shadow memory is updated instantaneously. All RTL writes are also communicated to shadow memory in a similar fashion. A very minimal change is required on ESL simulator to make available the just-in-time shadow memory support, which can be quickly done with the help of provided examples.

All the write updates are communicated to either side to maintain an identical copy on both sides. Hence, the shadow memory and RTL memory are kept cycle-wise consistent. This considerably reduces the time to view the memory, to enable debug read/writes, and helps providing better debugging support on the ESL simulator through a well-defined debug interface.

[1] Just-in-time shadow memory technique is filed for patent.



Accumulating data buffer with signal changes
Running
RTL clock rises
write to socket (blocked)
New HDL Signal Values
socket read (blocked)
wait
Blocked
socket read served
New Transactor's Signal Values
Running
Update signal values (XTOR-mapped singals)
HDL Simulator
socket read (blocked)
cycle started
wait
Blocked
socket read served
Update signal values (XTOR)
Running
transactor's signal value changes are accumulated on a data buffer
wait
write to socket (blocked)
cycle finished
The ESL Simulator

Figure 2: Synchronization between the ESL simulator and logic simulators. The light grey part shows when the simulator is blocked for a read/write operation, whereas the dark grey region shows the running status of the simulators.

3.2 Synchronization/Communication

In this subsection, we talk about the synchronization and communication details.

3.2.1 Synchronization

In our implementation, all events taking place between two clock cycle edges are abstracted to one point during the cycle. At this point, two simulation interface functions, *communicate* and *update* are called [15]. In *communicate* function, all inter-component communication is done as the name implies. In *update* function, shared resources are updated while no communication takes place. The main scheduler calls these two functions of each component that comprises a system while emulating concurrency of the hardware components of the system. For cosimulation, we have chosen the rising edge of a clock cycle for synchronization between the ESL simulator and the logic simulator. At the rising edge, the two functions, *communicate* and *update* are both called before returning control to the logic simulator.

Figure 2 illustrates how the ESL simulator and a logic simulator are synchronized while communicating for changes in signal values. There are four important events involved with the synchronization:

1) ***Beginning of the ESL simulator cycle***

The ESL simulator waits on a blocked read. The logic simulator executes and upon next rising edge of the clock it will send data via socket. The data arriving at the socket releases the blocked read on the ESL simulator side. All updates on signals in HDL simulation are sampled by the ESL simulator.

2) & 3) ***Communicate and Update***

The ESL simulator's cycle-based computation is done in all models existing in the simulator. Changes in all the signals are accumulated in a data buffer while the HDL simulator is waiting on a blocking read.

4) ***End of the ESL simulator cycle***

Within the callback for this event, changes in all the signals are sent to the HDL simulator via a socket. (This releases the HDL simulator's blocking read).

The HDL simulator samples the changes on signal values sent by the ESL simulator and applies them to the corresponding signal objects. Once the HDL simulator is free, it computes the changes on the signals connected to the transactor and accumulates them in a buffer for next event. The ESL simulator is also let go free and the callback for the event of the beginning of ESL simulator cycle will be called incurring the next cycle's wait on blocking read. The same entire process is continued for every cycle for both simulators.

3.2.2 Communication

As described in the previous subsection, at the synchronization points we need to transfer the changes in all exported signals using a communication primitive. Communication layer between the ESL simulator and RTL simulator plays a key role in overall performance and cycle-accurate synchronization. In our implementation, we generate a proxy module (in HDL) which specifies all the exported signals from the software side to the hardware side. All the master signals are driven from the ESL simulator's side and all the slave signals are driven from the RTL side. The communication data is in the form of driving these signals on either side.

All communications between the system components on ESL simulator side are in the form of transactions or signals. But RTL side does not understand the transaction-level modeling. Hence, all the transactions to the RTL side must be converted into signals. A transactor that converts a transaction to a set of signals is placed in the ESL system design to export required signals. The transactor is responsible for driving and reading all the exported signals accurately. The implementation of the converter is based on the protocol of the transaction and should be implemented by the system designer.

The RTL layer library uses VPI/PLI accesses in case of Verilog and FLI/VHPI accesses in case of VHDL to access all the exported signals in RTL simulation. We find that our cosimulation speed is mainly limited by the speed of RTL simulation. Therefore, we try to minimize the number of PLI/VPI/FLI accesses to the RTL code as much as possible. There is also some delay overhead associated for communicating the data from RTL side to our side and *vice versa* on each cycle. We try to minimize the amount of data to be communicated by sending only the value changes in signals rather than the values of all the exported signals in each cycle.

All the data exchange between the simulator and logic simulators is done using file-based socket in Unix-based systems. While in window platform, a TCP/IP socket is used to transfer the data. Although TCP/IP based socket is a solution for remote cosimulation, but remote cosimulation is generally less preferred. For local cosimulation, file-based socket communication provides good performance on Unix-based systems because it does not incur the TCP/IP protocol overhead. System V IPC in Unix-based systems provides communication primitives such as shared memory, semaphores, and message queue. We compare the performance of cosimulation by implementing these communication primitives, which we discuss next.

4. Evaluating Communication Primitives

In this section, we present evaluation of communication backplane between the hardware and software simulators. Since the communication and synchronization overhead affects the overall speed of simulation, there is a pressing need to realistically quantify the cosimulation performance considering different communication primitives. We implement three communication primitives available in Linux system: 1) shared memory 2) message queue and 3) file-based socket. Since shared memory does not come with its own synchronization, we had to implement the synchronization part of the shared memory using semaphores. Due to this extra overhead, we found that cosimulation performance is almost more than hundred times worse than message queue or file-based socket implementation. So we need to find better synchronization mechanism for shared memory and we concentrate on message queue and file-based socket for the rest of our analysis.

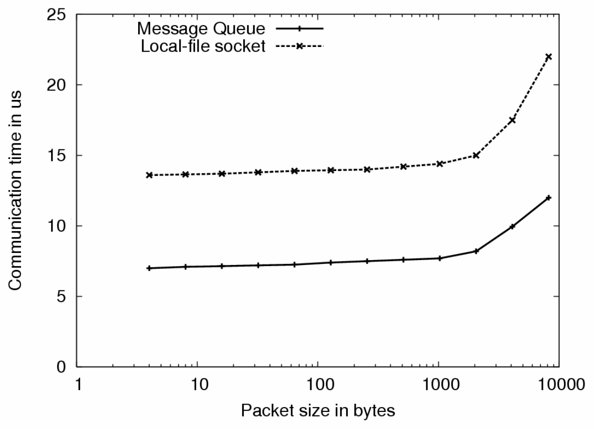


Figure 3: Evaluating communication primitives by varying packet size. The communication time overhead is shown on y-axis and packet size is shown on the x-axis in log scale.

We implement message queue communication routines and use the built-in features for synchronization. The main limitation in message queue is that the maximum size of the packet supported is 8192 bytes. Hence, to send larger size packets, it has divided into many chunks before communicating. We implement the file-based socket in the same way as message queue and the maximum size of the packet supported is 64 Kbytes.

4.1 Changing Message Size

We first evaluate the message queue and file-based socket primitives using a standalone environment where we change the size of the message. We communicate a million packets using both primitives and record the time of execution and using this time, we calculate the time of communicating a message of particular size for both primitives. The communication time for different sizes of messages (for both primitives) is shown in Figure 3. The message size is shown on the x-axis in log scale, whereas y-axis shows the communication time in µs. As we can see that increase in communication time is less than 7% when we increase the message size up to 1024 bytes. But as we increase the message size from 1024 to 8192 bytes, we see a sharp increase in the communication overhead. In the case of message queue, the communication time increases about 20% when we increase the message size from 4kbytes to 8Kbytes. For the same size increase in file-based socket, the percentage increase in communication time is found to be 25%. When we compare both message queue and local socket, we find that message queue provides a significant lower communication overhead for all the message sizes. For example, to communicate a message of

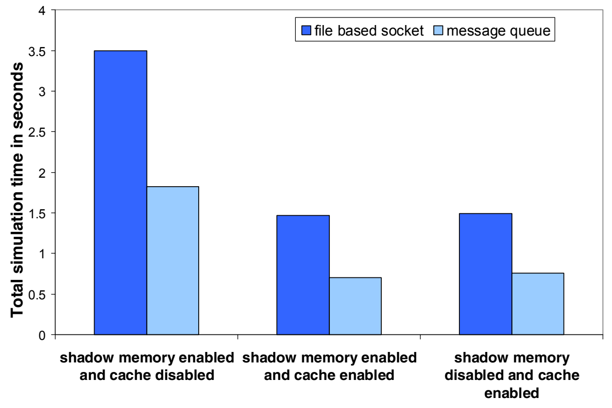


Figure 4: Communication/Synchronization overhead for a real system with ARM926EJ-S processor with RTL memory. Three configurations are compared as shown. Message queue based implementation is shown to perform better in all the three cases,

4kbytes, local socket based implementation requires more than two times of the communication overhead in message queue based implementation.

4.2 Evaluating ARM926EJ-S Based System

We evaluate both the communication primitives using an ARM926EJ-S system with RTL memory. We consider three scenarios: 1) When shadow memory is enabled and cache is disabled 2) Shadow memory is enabled and cache is enabled 3) shadow memory is disabled and cache is disabled. We measure the cosimulation performance in these three different scenarios for both message queue and file-based socket and the results are presented in Figure 4. When cache is disabled, the memory traffic is much higher and it involves much higher communication overhead as shown in the figure. We find that message queue implementation provides almost 50% more performance compared to file-based socket implementation for communication backplane. This is true for all three scenarios. From Figure 4, we can also see that we get slight increase in cosimulation performance when shadow memory is enabled (comparing scenario 2 and 3). For all the three scenarios, communication overhead is two times less in message queue based communication platform than that compared to local file based socket. Hence, message queue based communication backplane can provide significant performance advantage compared to a local file based socket implementation.

5. Conclusion

Hardware-software cosimulation is becoming more popular because of simulation speed and its usefulness in co-verification space. We addressed some of the key challenges associated with cosimulation including debugging support and communication overhead. We presented the just-in-time shadow memory technique that provides RTL memory view, debug read/write accesses to RTL memory and better debugging capabilities on the ESL simulator side. The synchronization issues along with our implementation details between an RTL simulator and an ESL simulator was also described. We also presented a study of communication backplane implemented using different communication primitives. We find that message queue is a better solution for communication backplane instead of file-based sockets when cosimulation performance is evaluated for a real system design. In the future, we plan to investigate the communication primitives for cosimulation using a multi-core system design.

References

[1] Kyuseok Kim, Yongjoo Kim, Youngsoo Shin and Kiyoung Choi, “An integrated hardware-software cosimulation environment with automated interface generation”, *Seventh IEEE International Workshop on Rapid System Prototyping*, pp. 66 – 71, June 1996.

[2] William D. Bishop and Wayne M. Loucks, "A Heterogeneous Environment for Hardware / Software Cosimulation," *in the Proceedings of the 30th Annual Simulation Symposium,* pp. 14-22, Atlanta, Georgia, April 1997.

[3] M. Lajolo, A. Raghunathan, S. Dey, L. Lavagno, “Cosimulation-Based Power Estimation for System-on-Chip Design”, *IEEE Transactions on VLSI Systems*, Vol. 10, No. 3, pp. 253-266, June 2002.

[4] C. Lennard and D. Mista. "Taking Design to the System Level", *ARM White paper*, April 2005.

[5] S. Lee, S. Das, V. Bertacco, T. Austin, D. Blaauw, and T. Mudge, .Circuit-Aware Architectural Simulation,. *in the 41st Design Automation Conference (DAC-2004),* June 2004.

[6] D. Becker, R. K. Singh, and S. G. Tell. “An engineering environment for hardware/software co-simulation”. *In Readings in Hardware/Software Co-Design,* Kluwer Academic Publishers, Norwell, MA, 550-555.

[7] C. Passerone, L. Lavagno, M. Chiodo, and A. Sangiovanni-Vincentelli, "Hardware/Software Co-Simulation for Virtual Prototyping and Trade-off Analysis", in *Proceedings of the 34th Design Automation Conference (DAC'97),* Anaheim, California, USA, June 9-13, 1997, pp. 389-394.

[8] J. Rowson. "Hardware/Software Co-Simulation," *Design Automation Conference Proceedings,* June 1994, pg 439.

[9] D. Kim, Y. Yi and S. Ha. "Trace-Driven HW/SW Cosimulation Using Virtual Synchronization Technique", *Design Automation Conference Proceedings* June 13-17 2005

[10] L. Séméria and A. Ghosh. "Methodology for hardware/software co-verification in C/C++". *In ASP-DAC* 2000: pages 405-408.

[11] C. Liem, F. Naçabal, C. A. Valderrama, P. G. Paulin, and A. A. Jerraya. "System-on-a-Chip Cosimulation and Compilation". *IEEE Design & Test of Computers* 14(2), pages 16-25, 1997.

[12] W. Sung and S. Ha. "Optimized Timed Hardware Software Cosimulation without Roll-back". *In DATE* 1998, pages 945-946.

[13] V. zivojnovic and H. Meyr. "Compiled HW/SW co-simulation". *In Proceedings of the 33rd Annual Conference on Design Automation*, Las Vegas, Nevada, United States, June 03 - 07, 1996.

[14] B. Bailey, R. Klein, S. leef. "Hardware/software Co-Simulation Strategies for the future", White paper.

[15] ARM Ltd., "Cycle-Accurate Simulation Interface (CASI) – RealView ESL API" (http://www.arm.com/products/DevTools/RealViewESLAPIs.htm