A Hybrid Channel for Co-Simulation of Behavioral SystemC IP with its Full System Prototype on FPGA*

Antonis Papagrigoriou, TEI of Crete, Heraklion, Greece (apapa@cs.teicrete.gr)

Miltos D. Grammatikakis, TEI of Crete, Heraklion, Greece (mdgramma@cs.teicrete.gr)

Voula Piperaki, TEI of Crete, Heraklion, Greece (vpiperaki@cs.teicrete.gr)

***Abstract*— This is a report on experience, using a bidirectional asynchronous channel between a SystemC device model and a physical, full system prototype of the same device on FPGA. The asynchronous channel uses TCP sockets when the SystemC model is on a different host, and shared memory when the model is on the same host. Synchronization is maintained between the two sides by aligning the SystemC side with the real-time clock (using Realtimify). The full system is used to co-simulate a complete firewall application, consisting of a NoC firewall module (the DUT), drivers, OS, and application software running on an ARM v7 Zedboard. The asynchronous channel has good performance, especially for shared memory communication, with an overhead in the ms range. Overall simulation speed is sufficient such that real-time performance characteristics can be verified.**

***Keywords*— ARM; co-design; co-simulation; co-validation; FPGA; high-level synthesis; SystemC; TLM**

I. Introduction

As shown in Figure 1 (a), traditional SystemC — RTL co-validation techniques are based on co-simulation of **RTL models** of the design under test (DUT) with the corresponding system-level model usually implemented in a **virtual platform** [3, 4, 12]. Thus, in this design flow, an application scoreboard can simultaneously compare the input interface signals on the time-annotated SystemC model and the synthesizable RTL reference model, and provide assertions (e.g. in UVM [1]) that compare the corresponding traces for specific application testbenches.

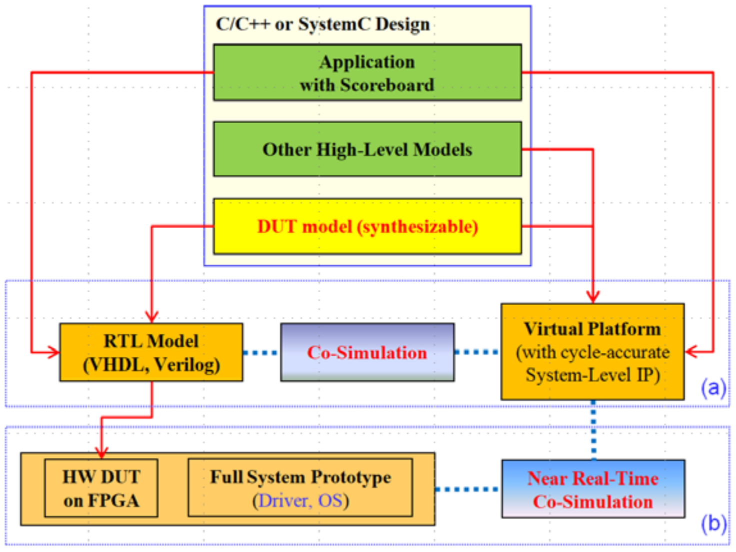

C/C++ or SystemC Design
Application with Scoreboard
Other High-Level Models
DUT model (synthesizable)
RTL Model (VHDL, Verilog)
Co-Simulation
Virtual Platform (with cycle-accurate System-Level IP)
(a)
HW DUT on FPGA
Full System Prototype (Driver, OS)
Near Real-Time Co-Simulation
(b)

Figure 1. Traditional RTL co-validation (shown in a) vs proposed near real-time full system co-validation technique (shown in b).

In contrast, as shown in Figure 1 (b), our technique focuses on validating a full system, by performing near real-time, *command-to-command co-simulation* of the **hardware DUT implemented on FPGA as a full system prototype** (with CPU, on-chip interconnect, memory, application, drivers, and operating system, typically Linux)

* Work funded by National Matching Funds 2017-18 of Greek Govt. (GSRT) related to EU project "FP7-DREAMS" (GA 610540).

against its equivalent cycle-approximate system-level model of the DUT. Notice that *command-to-command co-simulation* means that an application command (e.g. memory-mapped to HW DUT on FPGA) is run on both the hardware DUT and its equivalent SystemC model before any subsequent application command is executed. Thus, in our approach, we keep the hardware DUT and system-level IP synchronized and check for a specific application testbench, command-for-command, if transformations during high-level synthesis preserve correctness and satisfy specifications, such as performance requirements.

This concept is particularly important for co-validation during rapid prototyping. For example, we can validate correct translation by the high-level synthesis tools.

II. Co-Simulation Platform

In order to support *command-to-command co-simulation*, an asynchronous channel is necessary for efficient interaction between real and virtual world (and vice-versa). This hybrid channel will allow a scoreboard to closely monitor and validate changes in control flow or system state. For example, simulation output from the virtual world may result, via OS-layer interactions, in updating parameters or data structures in the real world. This can occur at discrete time intervals, e.g. during SystemC evaluate cycle, update cycle, or at a later time synchronization point. In a similar pattern, events captured in the real world, such as data output or system interrupts, can be relayed to the virtual world, as SystemC events.

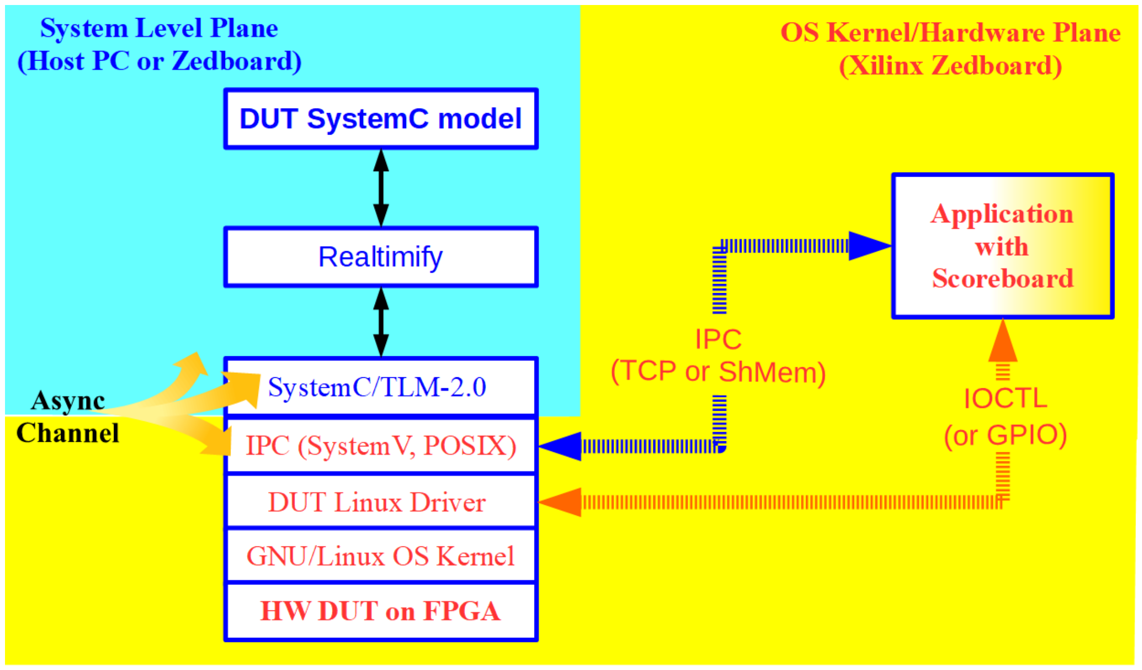


Figure 2. Layered co-simulation approach, showing OS Kernel/Hardware Plane (real world) and System Level Plane (virtual world).

Figure 2 provides a general perspective on the proposed near real-time co-simulation platform and introduces the asynchronous, bidirectional, hybrid SystemC channel (called asynchronous channel).

- The bottom and top right part (called *OS Kernel/Hardware Plane*, or *real world*) implements **IOCTL or GPIO communication** for accessing the memory-mapped HW DUT device via its Linux driver from an application scoreboard. This software mechanism resides in the platform on which the HW DUT is embedded, typically a prototype FPGA development board running Linux; in our case, a Xilinx Zedboard.
- The top left part (called *System Level Plane*, or *virtual world*) concerns the system-level model which communicates with the external application scoreboard via an asynchronous, bidirectional, hybrid SystemC channel. This channel implements IPC (either TCP socket-based, or shared memory mechanisms) to interact during runtime with the application scoreboard. The channel a) propagates each command received from the application scoreboard to the SystemC DUT model, using an asynchronous TLM socket, b) activates the SystemC model, possibly including a so-called *Realtimify* module [9] for

synchronizing the real clock of the Hardware DUT with the clock of our SystemC kernel and c) returns, through another TLM socket, timing, status and performance characteristics from the system-level IP (running on host PC or Zedboard) back to the application scoreboard for co-validation against output from the hardware DUT (running on Zedboard). The application scoreboard may invoke standard UVM technology to perform co-validation, cf. blue and red arrow paths. The system-level IP could run either on a host PC running Linux on top of the SystemC kernel, or even on Zedboard, i.e. the system on which the hardware DUT is prototyped.

Our co-simulation platform uses the Realtimify SystemC module to synchronize execution of the SystemC model with respect to real-time, e.g. using *usleep()*. The synchronization interval in Realtimify must be carefully selected to match requirements of the application and complexity of the hardware DUT. If this interval is not carefully selected, there could be an increase in the duration of the SystemC evaluate cycle when modeling the System-Level IP, and as a result we would not be able to verify the hardware DUT in near real-time.

As explained above, one of the relatively novel techniques that our co-simulation framework defines is an asynchronous, bidirectional, hybrid SystemC channel for connecting a simulated device with its equivalent full system prototype. This allows asynchronous co-validation of the real system with the simulated device in near real-time. The proposed asynchronous channel extends preliminary ideas by Black presented in DVCon 2014 that advocate using TCP sockets for communication between virtual world (SystemC model) and real world (hardware DUT embedded on FPGA in an embedded system running Linux) [2, 3]. Specifically, we have implemented a bidirectional socket- or shared memory-based channel interface from the Hardware Plane to the System Level Plane.

Our co-simulation environment

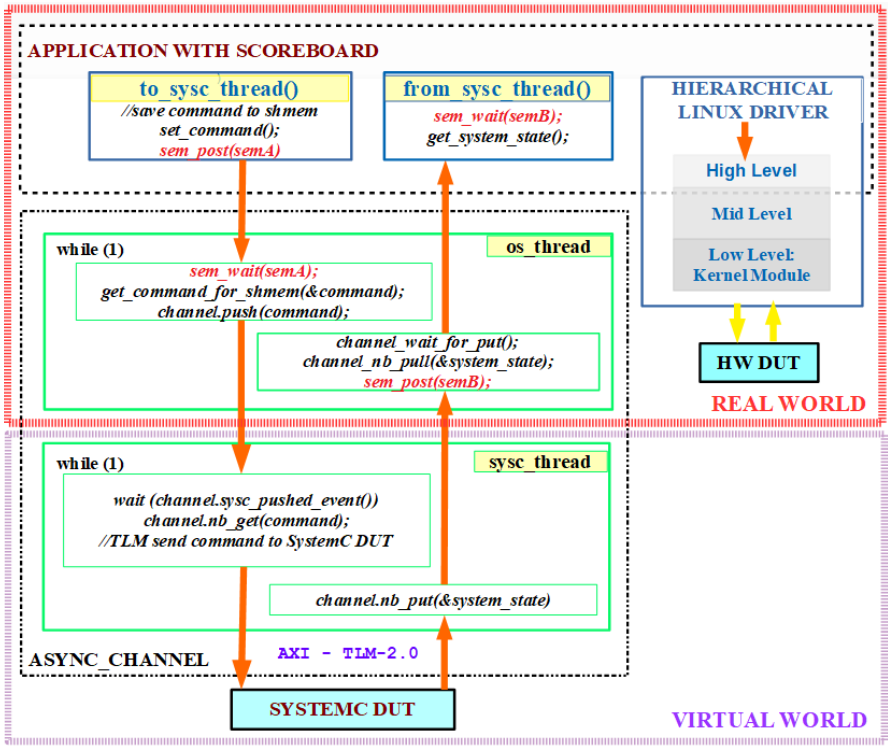
- provides an **open-source experimental framework for co-validating a hardware DUT (embedded in a real system with an OS stack) against its equivalent SystemC model,**
- implements **a non-intrusive asynchronous SystemC channel based on shared memory or TCP socket primitives**. This channel allows an application scoreboard to send commands to a SystemC model for co-validation against a full system prototype device (running in an FPGA); this device is accessed by the scoreboard via IOCTL (or GPIO).
- **(optionally) supports a near real-time option using Realtimify SystemC module that allows matching simulation time with real-time execution.**

The asynchronous, hybrid channel that we have used is a relatively novel approach for co-simulating SystemC models with real systems [5]. Alternative full system validation techniques, such as in-circuit prototyping from inPA systems in [6], stimulate the system via its real-world interfaces, meaning that validation is executed at real-time speed, but ignore comparisons with equivalent executable specifications in system-level design languages. Another closed-source industrial approach extends the Standard Co-Emulation API: Modeling Interface (SCE-MI) developed by Accellera for SystemC and RTL co-simulation, to address faster SoC simulation between SystemC and FPGA prototypes, but does not address a full system built around the FPGA prototype [6]. Similarly, commercial rapid prototyping tools from Synopsys allow a virtual prototype (ARM Versatile Express running Linux) to directly perform transactions on an FPGA-based prototype (peripheral subsystem synthesized from HDL source) via two channels: a) a signal-level link for cycle-accurate co-simulation, and b) a transactor library based on TLM-2.0 for efficient transaction-based validation [10]. The transactor library API models communicate to SoC models and real blocks (assigned to FPGA-based prototype) using the AMBA bus: read, write, and callback operations (e.g., see testbench in Section 3). Other commercial SystemC/FPGA prototyping solutions address full system co-validation **only on the side of the virtual platform**, e.g. consider ARM Fast Models, SystemC with QEMU, and Cadence hw/sw codesign [7].Compared to these tools, the proposed platform expands the scope of co-validating a hardware DUT prototype built on FPGA **in a real system with OS stack against its equivalent SystemC model** for the first time. Past work has considered the hardware DUT in the context of bare metal or within fuly virtualized systems [7] [10].

III. THE ASYNCHRONOUS CHANNEL

Figure 3 illustrates the block diagram of the co-simulation platform, outlining how an application scoreboard can communicate commands to the SystemC model (System-Level Plane) using the proposed asynchronous channel. The channel is implemented using shared memory (or simply using TCP sockets). The application can also communicate with the memory-mapped HW prototype device, the DUT, (OS Kernel/Hardware Plane) on the FPGA development board, via its hierarchical Linux driver. Finally the application can receive and compare respective responses from both the System model and Hardware DUT.

We next focus on the asynchronous channel, examining its data structures and methods. In order to communicate with the virtual world (System-Level Plane), the application scoreboard defines two POSIX threads: a) *to_sysc_thread* which transfers command packets from the OS to our asynchronous channel, and b) *from_sysc_thread* which transfers results in the opposite direction. These two POSIX threads in the application scoreboard communicate with a **POSIX thread in our asynchronous channel** (*os_thread*) implemented in the asynchronous channel using POSIX shared memory and POSIX semaphores, as shown in Figure 3; (alternatively, they can use TCP sockets alone, or utilize SystemV shared memory and POSIX semaphores if they are defined as processes).



```
//save command to shmem
set_command();
sem_post(semA)
```

```
sem_wait(semB);
get_system_state();
```

```
while (1)
    sem_wait(semA);
    get_command_for_shmem(&command);
    channel.push(command);

    channel_wait_for_put();
    channel_nb_pull(&system_state);
    sem_post(semB);
```

```
while (1)
    wait (channel.sysc_pushed_event())
    channel.nb_get(command);
    //TLM send command to SystemC DUT

    channel.nb_put(&system_state)
```

Figure 3. Block diagram of the asynchronous channel implemented within our Co-Simulation Platform.

In turn, the *os_thread* synchronizes with a **second SystemC thread in our asynchronous channel** (*sysc_thread*). The *os_thread* and *sysc_thread* support the necessary channel interface primitives (asynchronous blocking *push/pull* functions and non-blocking *nb_get*/*nb_put* methods) to transfer commands and results. A command, eventually destined for the SystemC DUT, will be temporarily buffered in the *queue_to_sysc, and read* by the SystemC DUT input thread via its *TLM-2.0 AXI Interface* (see below). Subsequently, as a result of executing the command, result data from the SystemC DUT, eventually destined for the scoreboard (in the *os_thread*), will be temporarily buffered in the *queue_from_sysc* and finally read by the *from_sysc_thread*.

These channel primitives are protected by mutexes (*m_mutex_to_sysc and m_mutex_fm_sysc*) and are shown in the code snippet below.

```
// os_thread methods for communicating to/from sysc_thread
void push (async_packet async_payload_ptr);
void nb_push (async_packet async_payload_ptr);
bool can_push (void);
void pull (async_packet &async_payload_ptr);
bool nb_pull (async_packet &async_payload_ptr);
bool can_pull (void);
```

```
// sysc_thread methods for communicating to/from os_thread
void nb_put (async_packet async_payload_ptr);
void put (async_packet async_payload_ptr);
bool can_get (void)
bool nb_get (async_packet &async_payload_ptr);
void get (async_packet &async_payload_ptr);
bool can_pull (void)
// sysc_thread events
const sc_core::sc_event& sysc_pushed_event(void)
const sc_core::sc_event& sysc_pulled_event(void)
```

The command packet data structure in the asynchronous channel is listed below:

```
typedef struct {
  int32_t  tid;      // unique transaction id for each packet exchanged
  uint8_t  command;  // for memory-mapped devices, AMBA AXI read/write or interrupt
  uint8_t  status;   // SystemC TLM socket communication status
  uint32_t address;  // physical address to access device (e.g. memory-mapped)
  uint8_t* data_ptr; // result data
  uint16_t data_len; // result data length
  uint64_t rx_timestamp; // timestamp upon receiving the command packet
  uint64_t tx_timestamp; // timestamp placed upon transmitting the result
} async_packet;
```

Since we mainly target ARM-based reconfigurable embedded systems (i.e. we use an AMBA AXI System Interconnect for prototyping on the Zedboard), we implement, (cf. bottom of Figure 3), a limited *TLM*-2.0 *AXI interface* between the asynchronous channel and the System-Level IP. Using this interface, the system-Level IP of the DUT can respond to external memory read/write or interrupt operations arriving via the asynchronous channel. For a list of commands used in our testbenches, refer to Section IV.

The interface is implemented by extending the SystemC TLM template class *simple_initiator_socket<T>* with the methods and states to operate as a master AXI interface and support 4-way AXI handshaking with normal read/write and burst read/write transactions. We similarly extend the SystemC TLM template class *simple_target_socket<T>* to operate as an AXI-Lite Slave interface and support 4-way AXI handshaking with only normal read/write transactions.

The final module (omitted from Figure 3) is a so-called *Realtimify module* [9]. Using this module, we are able to synchronize the clock of our SystemC kernel with the real clock of the Zedboard running the application scoreboard. Notice also that the system frequency and CPU power affect the simulation clock. By using the Realtimify module alongside the blocking methods of the asynchronous channel, the System-Level IP can react to external events during runtime.

IV. NoC Firewall DUT and Application Testbenches

Preliminary evaluation is based on prototyping a NoC Firewall targeting data privacy in m-Health applications. The hardware module (DUT) has been designed in cycle- and bit-accurate SystemC and synthesized on the ARM v7 Zedboard using Xilinx Vivado HLS and Vivado tool v2014.4.1 [11]. As shown in Figure 4, the hardware DUT, supported by a *hierarchical Linux driver*, attempts to thwart on-chip access to different output ports (BRAMs) by setting read/write deny rules that take into account not only the BRAM number (1 to 4), but also the actual network access path.

Our bit-accurate SystemC model of the NoC Firewall model consists of 1700 lines of code, with 14 SystemC clocked threads (most of them related to module interfaces). Vivado NoC Firewall synthesis on Zedboard FPGA uses: 11421 LUTS, 12012 registers and 4 BRAMs. In comparison, a 12-node STNoC from STMicro configured with 4x4 routers uses: 24939 LUTS and 17983 registers on Zedboard FPGA. The hierarchical Linux driver is approximately 1900 lines of code, and the application (including crypto library calls) approximately 2500 lines of code. For more details and downloading the open hardware and software for installing on Zedboard, cf. [8].

Although details are omitted due to space restrictions, it's sufficient to say that the firewall setup corresponds to writing a register set configured to protect data in a particular BRAM address range from illegitimate access from specific input ports. This NoC Firewall setup info is always checked before any read/write access to a BRAM. Setup and access functions are supported by a hierarchical driver that performs memory-mapped operations at admin and user level via IOCTL.

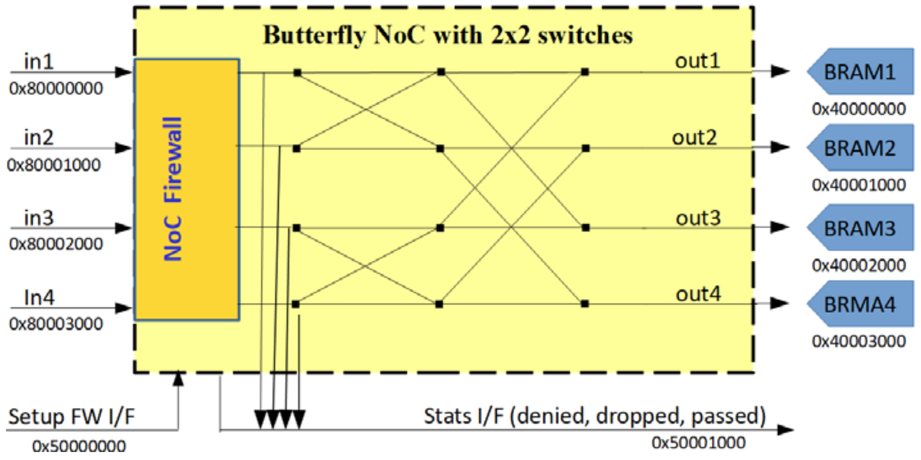


Figure 4. The NoC Firewall DUT.

In our Linux driver implementation, we handle firewall setups for specific input and output ports ranges by accessing memory-mapped firewall setup registers from the kernel space via **Low-Level Driver** (LLD) I/O memory accesses: ioread32 and iowrite32 operations. In addition, the **Mid-Level Driver API** (MLD) defines IOCTL calls for firewall setup and read/write access to BRAMS and statistics registers (logs the number of passed, dropped, denied packets) from specific input and output ports on the NoC Firewall. This involves a) AXI-to-NoC packetization of the write access request at the NoC interface, as well as transfer of the NoC packet for execution at the BRAM and b) AXI-to-NoC packetization of the read request with subsequent response NoC-to-AXI de-packetization for read data. The **High-Level Driver** API (HLD) completes the driver hierarchy by supporting high-level data privacy and security primitives for industrial m-Health applications [8].

For accessing the SystemC model via the asynchronous channel, we use the same philosophy, and implement counterpart **sc_ioread32/sc_iowrite32 operations** (equivalent to LLD's ioread32/iowrite32), and connect them with the the Mid-Level Driver API.

In this context, three different testbenches are considered:

Our **first two DUT testbenches** provide complete coverage tests for a large matrix of test conditions: co-validating **read** and **write** access to Rule, Statistics registers and BRAMs via the NoC Firewall for all possible input and output ports, for all possible settings of the firewall (accept all, deny read, or deny write mode for each port), and for both Simple and Extended Operating Mode. The tests also resolve synchronization issues related to accessing BRAM via two different paths: through the NoC firewall and direct access via another AXI interface (backdoor). The **final testbench (mHealth)** concerns a modular NoC firmware security solution that supports two modes. First, it provides **anonymity** of patient data by hashing to an appropriate location in FPGA memory (BRAM) using the unique patient ID. Second, it provides **healthcare data protection** in BRAMs from malicious or unauthorized physicians (illegitimate Linux group ID) by setting NoC Firewall rules and controlling network paths to BRAMs based on group ID.

V. Experimental Framework and Co-Simulation Results

We first consider average application delay for our different channel implementations, considering all testbenches: full coverage read test via the firewall, similar write test, and telemedicine (m-Health) application supporting secure access to patient data based on Linux group.

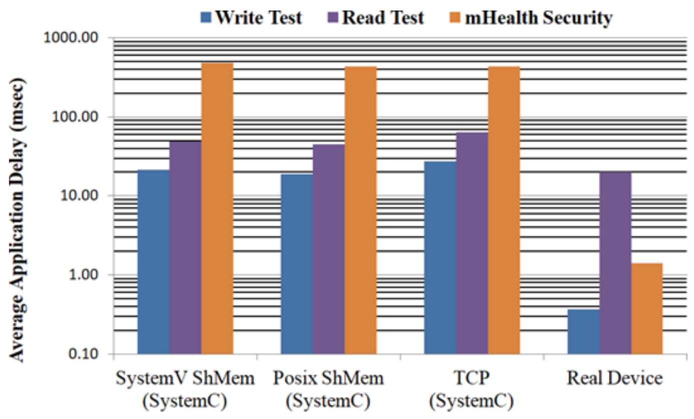


Figure 5. Average testbench delay for a) SystemC model (with POSIX, SystemV shared memory or TCP-based asynchronous channel) and b) the real device (full system running Linux) for all three applications on Zedboard.

In Figure 5, we compare average delay for running each testbench 100 times, i.e. accessing the NoC Firewall HW DUT built on Zedboard FPGA via hierarchical Linux driver, and its equivalent SystemC model via our asynchronous channel. Our asynchronous channel implementations provide different performance metrics: POSIX shared memory is 6.5 to 11.1% faster than SystemV, and 22 to 23% better than TCP. The latter TCP version is less efficient, although we have used the loopback interface (localhost) to minimize communication delay. The loopback interface will consume space in the Linux kernel socket buffer until the packet is processed, just as if it had come from the Ethernet while avoiding real Ethernet delays. Real Ethernet layer delay could be important for supporting remote co-simulation, e.g. similar to Amazon Web Services for FPGA access.

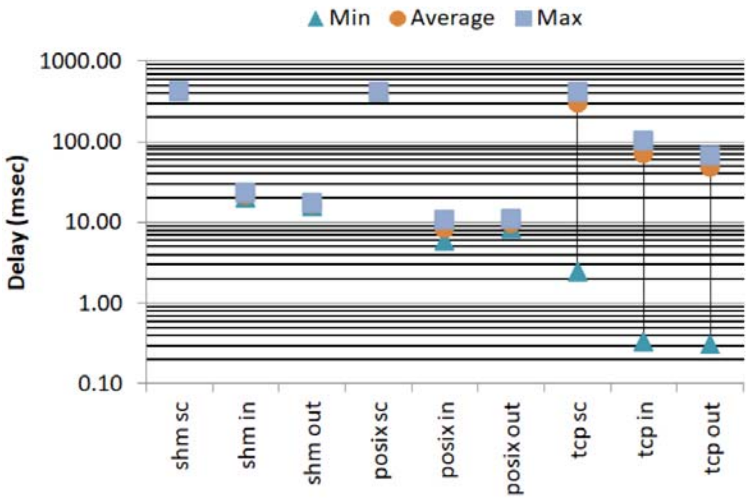


Figure 6. Asynchronous channel and SystemC model delay for POSIX-, SystemV shared memory-, and TCP-based implementation, assuming mHealth application on Zedboard.

In Figure 6, we consider the mHealth security application on Zedboard and compare the minimum, average and maximum delay for each communication stage in our asynchronous channel using system timestamps. Notice three types of delay (in ms): ***_sc** values correspond to SystemC model delay, ***_in** values relate to overhead for channel input, and ***_out** values are the delays for channel output. Our channel is non-intrusive, since its delay is small relative to SystemC application delay. For POSIX shared memory only 5.2% of the total time is spent for channel input and output compared to SystemC processing; comparably, this overhead is 5.6% for SystemV, and larger, up to 41.0% (with very large fluctuation) for TCP socket implementation. For the SystemC testbench code, refer to [8].

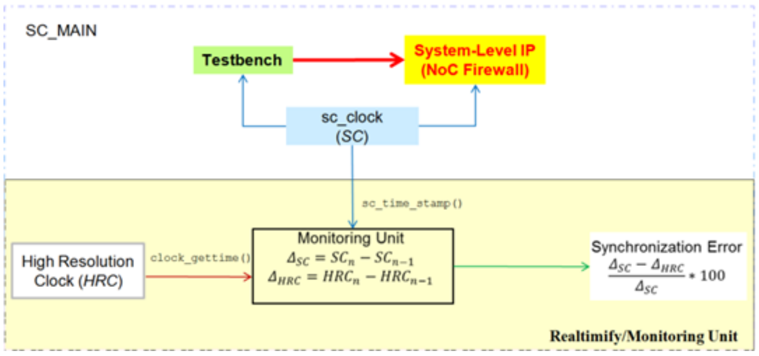

SC_MAIN
Testbench
System-Level IP (NoC Firewall)
sc_clock (SC)
sc_time_stamp()
High Resolution Clock (HRC)
clock_gettime()
Monitoring Unit
$\Delta_{SC} = SC_n - SC_{n-1}$
$\Delta_{HRC} = HRC_n - HRC_{n-1}$
Synchronization Error
$\frac{\Delta_{SC} - \Delta_{HRC}}{\Delta_{SC}} * 100$
Realtimify/Monitoring Unit

Figure 7. Experimental framework focusing on Realtimify/Monitoring Unit.

Next, we focus on channel overhead with Realtimify, examining first the synchronization error for using this module during co-simulation. Our experimental framework focuses on measuring the System-Level Plane overhead, examining the synchronization error for our NoC Firewall model. For this reason, as shown in Figure 7, we record at regular application–specific intervals the current simulation time (*sc_time_stamp())* and compare against real-time (*clock_gettime()* with parameter *CLOCK_REALTIME*). The difference between these two timers refers to the time that simulation is ahead of real-time. Since this time difference must be zero for near real-time co-simulation, we propose either to suspend the model (e.g. using SystemC events), if model time is ahead, or freeze the real-time process (e.g. using system methods that allow entering an idle CPU state), if real-time is ahead of simulation time. Other techniques, such as using a simulation warp to the future, or selectively altering the abstraction level of the System-Level IP are not possible.

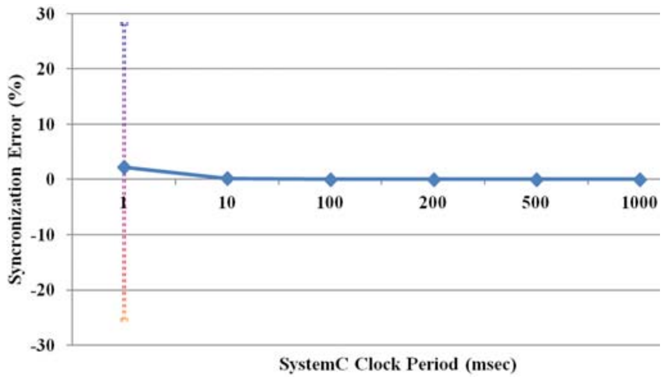


Figure 8. Average synchronization error on Zedboard.

As shown in Figure 8, for a SystemC period more than 1ms, we can support near real-time co-simulation between the SystemC model and the full system prototype in the ms period range, since the synchronization error is very small. This is especially true, if the system-level IP is placed on the board together with the DUT (communication via shared memory).

Figure 9 shows that with Realtimify, relative overhead in computation time of the proposed asynchronous channel is very small compared to the SystemC model (up to 3 orders of magnitude smaller), demonstrating efficiency of our channel for real-time co-simulation. Corresponding delays without Realtimify differ by only one order of magnitude.

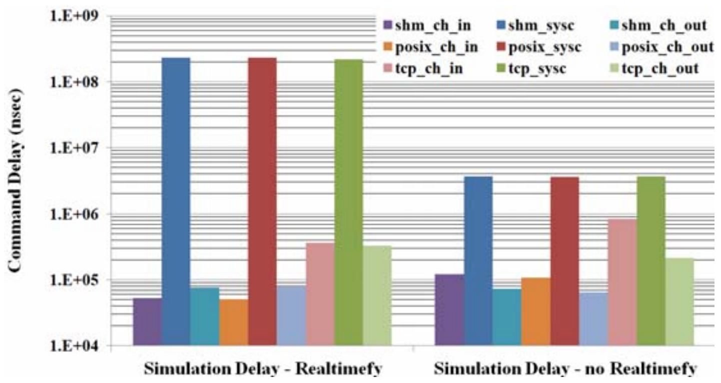


Figure 9. Asynchronous channel and SystemC model delays, assuming mHealth application on Zedboard.

VI. SUMMARY

We have developed a co-simulation platform, which enables co-validation of a hardware IP (DUT) prototyped as a full system on an FPGA development board, along with executable system-level specifications. We demonstrate efficiency of our platform, by considering an industrial m-Health security application based on a NoC Firewall IP. In the future, it would be interesting to examine near real-time co-simulation when the HW IP is embedded in a platform that uses a **real-time OS** (e.g. ThreadX), with appropriate drivers, and executes unmodified code (and not just memory accesses over AXI). By using the Realtimify module, the System-Level IP can react to runtime events (e.g. hardware interrupts from sensor devices) in a genuine environment.

ACKNOWLEDGMENT

We would like to thank Mr. D.C. Black of Doulos Corp. for sharing preliminary code which helped provide an incentive for developing further the "asynchronous channel" and co-simulation platform. The authors would also like to thank Mr. Mark Burton and the anonymous reviewers for valuable comments and suggestions that helped greatly improve the quality of our presentation.

REFERENCES

[1] M. Barnasconi, F. Pêcheux and T. Vörtler, "Advancing system-level verification using UVM in SystemC", Design and Verification Conference (DVCon), 2014

[2] D.C. Black. "A SystemC Technology Demonstrator" , SystemC Tutorial , Design and Verification Conf. (DVCon), 2013.

[3] D. C. Black, software code for SystemC technology demonstrator, available from https://github.com/dcblack/technology_demonstrator

[4] A. Ferrari, and A. Sangiovanni-Vincentelli, "System design: traditional concepts and new paradigms", in Proc. Int. Conf. Computer Design, 1999, Los Alamitos, CA, USA.

[5] D. Fennibay, A. Yurdakul, and A. Sen, "A heteregeneous simulation and modeling framework for automation systems", IEEE Trans. on CAD, 31 (11), 2012, pp. 1642—1655.

[6] C.-Y. Huang, Y.-F. Yin, C.-J. Hsu, T.B. Huang et. Al., "SoC HW/SW verification and validation", in Proc. IEEE Asia and South Pacific Design Automation Conf.. 2011, pp. 297—300.

[7] G, Martin, F. Schirrmeister, and Y. Watanabe, "Hardware/software codesign across many Cadence technologies", Handbook of Hardware/Software Codesign, S. Ha, J. Teich (eds.), Springer, 2016, pp. 1037—1070.

[8] V. Piperaki, A. Mouzakitis, M. Grammatikakis and A. Papagrigoirou, SystemC NoC Firewall, hierarchical Linux drivers, testbenches and co-simulation platform", https://github.com/angmouzakitis/student_xohw18-187

[9] Digital Force Project Realtimify, available from http://www.digital-force.net/projects/systemc_realtimify

[10] Synopsys, "Better software faster!", Ch6, "Hybrid prototyping and emulation", March 2015, available from http://www.synopsys.com/vpbook

[11] Xilinx, Vivado HLS, available from https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html

[12] W. Li, X. Zhang, H. Li, "Co-simulation platforms for co-design of networked control systems: An overview", Control Engineering Practice, 23, 2014 pp. 44—56.