Enabling TLM-2.0 Interface on QEMU and SystemC-based Virtual Platform

Tse-Chen Yeh, Zin-Yuan Lin, and Ming-Chao Chiang
Department of Computer Science and Engineering
National Sun Yat-sen University
70 Lienhai Road, Kaohsiung 80424, Taiwan, ROC
sdgp03@ms18.hinet.net, m983040031@student.nsysu.edu.tw, mcchiang@cse.nsysu.edu.tw

***Abstract*—This paper presents a QEMU and SystemC-based virtual platform that is capable of hardware modeling using TLM-2.0 interface. The proposed virtual platform is not only capable of running an operating system, but it is also capable of using such an interface to connect hardware models, such as the instruction set simulator to a bus model. We verify the functionality of such a platform by using it to boot up a full-fledged Linux while at the same time estimating its performance at the instruction-accurate level. Furthermore, TLM-2.0 interface makes our framework more compatible with other models using TLM-2.0 and more suitable for modeling at the early stage of ESL design flow.**

***Keywords*—TLM; bus functional model; QEMU; SystemC; virtual platform; ESL**

I. Introduction

In order to overcome the complexity of System-on-Chip (SoC), most of the Electronic System Level (ESL) [1] tools provide the virtual platform for co-simulating hardware/software at the early stage of the design flow. In order to unify the interfaces of modeling the transactions between diverse hardware models, TLM-2.0 standard [2] was proposed. The hardware/software emulation framework QEMU-SystemC [3], proposed in 2007 for the development of software and device drivers, has been upgraded to use TLM-2.0 as its modeling interface in 2009 [4]. Although the TLM-2.0 standard can assist to model the memory-mapped bus model, the transactions of the bus model are incomplete because it lacks transactions provided by the processor model. Another virtual platform [5], [6] that combines QEMU-SystemC with CoWare's Platform Architect was proposed in 2009. This framework utilizes lots of models provided by off-the-shelf Model Library [7] modeled with TLM-2.0 interface.

To show that the QEMU and SystemC based virtual platform we proposed is capable of the TLM-2.0 interface, we connect the instruction-accurate instruction set simulator (IA-ISS) with the direct memory access controller (DMAC) model using the simple bus modeled in TLM-2.0 interface. Certain adaptions on interfaces of simple bus are needed because some properties of transaction modeling are maintained in the QEMU internally.

II. Related Work

In this section, we begin with a brief introduction to SystemC and TLM-2.0 standard. Then, we turn our discussions to QEMU-SystemC and the QEMU and SystemC-based virtual platform we propose, which uses QEMU as the processor model.

A. TLM-2.0

TLM-2.0 is an modeling interface focused on modeling the memory-mapped bus model. It is capable of modeling from the temporal decoupling (TD) down to the approximately-timed (AT). However, modeling at the cycle-accurate level is beyond the scope of TLM-2.0. The capability of mixed level modeling is demonstrated by the simple bus, which is modeled using AT coding style, connected with diverse initiator and target models using TD, loosely-timed (LT), and AT coding style. We can demonstrate that our virtual platform is capable of TLM-2.0 by using the simple bus it provided to connect our ISS and DMAC model.

B. QEMU-SystemC

QEMU-SystemC [3] is an open source hardware/software emulation framework for the SoC development. It allows devices to be inserted into specific addresses of QEMU and communicates by means of the PCI/AMBA bus interface. Although the waveform of the AMBA on-chip bus of the QEMU-SystemC framework can be used to trace the operations performed on the slave device, no transactions about the processor and master device are provided by QEMU-SystemC or the TLM-2.0 appending version [4]. For instance, the instructions executed, the memory accessed, and so on, which are so valuable to the system designers, are unfortunately not provided.

C. QEMU and SystemC-based Virtual Platform

In order to provide all the information necessary for modeling the bus transactions, QEMU has to be converted from a virtual machine to an ISS that is capable of simulating all the hardware models down to the cycle-accurate level [8].

The virtual platform described herein is implemented as two threads running in a single process and communicating with a unidirectional FIFO, as shown in Fig. 1. In other words, the communication between QEMU and SystemC is one way so that the relative order of the instructions executed, the memory accessed, and the I/O write operations is retained by the packet receiver within the ISS wrapper and infrastructure interface. Thus, the interface can simulate different bus transactions for modeling the Bus Functional Model (BFM) by means

978-1-4244-9021-9/11/$26.00 ©2011 IEEE

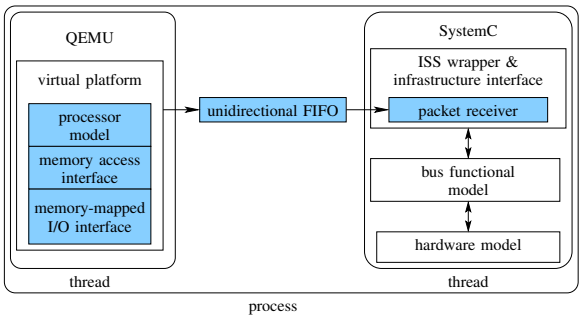
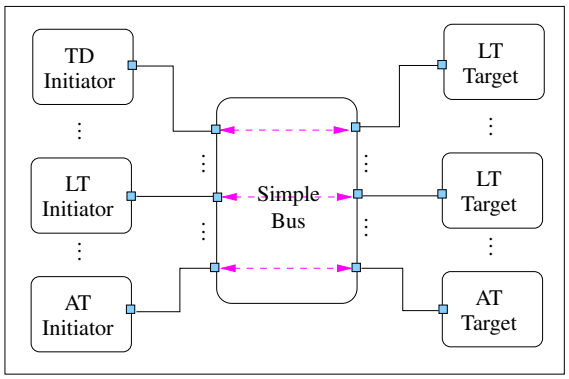


Figure 1. The inter-process communication (IPC) mechanism used by the ISS wrapper & infrastructure interface described herein.



Mixed Level Simple Bus Connections

Figure 2. Simple bus provided by TLM-2.0. All of the ports use simple sockets for connecting with each other. Also notice that the TD initiator is connected with LT target, and both of them are modeled by using the same LT coding style.

of the information in the received packets. In addition, the synchronization between QEMU and SystemC is only needed by the I/O read operations, which can be achieved by having QEMU call the I/O read function—which will pass the pointer to the data to be read to SystemC—and then block until SystemC returns.

III. Virtual Platform Integration

In this section, we give a more detailed description about how the simple bus is integrated into the virtual platform described herein. Because certain attributes of transaction modeling are maintained in QEMU, there is no need to re-model them.

A. Organization of Simple Bus

As described in Section II-A, the simple bus using AT coding style can be used to connect with different level of models. The simple bus is constructed by initiator and target sockets for connecting the external target and initiator models. The number of initiator and target sockets can be parameterized by using the template parameters at the time of model instantiation. All of the paths for propagating the transactions are similar no matter the blocking or non-blocking interface is used, as shown in Fig. 2. Furthermore, the mapped interface and utilities for modeling at different levels are summarized in Table I.

The simple bus is built on the base protocol pre-defined by TLM-2.0 instead of any specific bus protocol. The LT coding style supports two timing points for modeling the start and end of a single transaction, while the AT coding style provides four timing points for modeling the start and end of the request phase and the response phase.

B. Simple Bus Integration

In order to integrate the simple bus into our virtual platform, the first thing that has to be taken into consideration is what type of the simple socket is to be used. To model the connections between the ISS and DMAC model, we have to take into account four master ports, i.e., the processor instruction port, the processor data port, DMAC M1 master port, and DMAC M2 master port. Moreover, the processor data port can be divided into data port for the memory access and data port for the memory-mapped I/O access. For the simulation speed, we choose the LT style simple socket for modeling the initiator and target ports of our models.

As Fig. 2 shows, most of the initiator models are paired with the target models the propagating path built by the simple bus. However, two types of transactions initiated by the master ports need not to be transferred to the target models in our virtual platform. In this paper, we use the prefix "pseudo-" to indicate them, i.e., the pseudo-initiator, the pseudo-target, and the pseudo-transaction. The first type of the pseudo-transaction is initiated by the QEMU and launched by means of the pseudo-initiator socket of ISS wrapper. Moreover, the transaction sequence and the address translation are maintained by the transaction FIFO and the software MMU of QEMU, respectively.

The second type of the pseudo-transaction is initiated by the master ports of DMAC model and used to access the internal memory of QEMU. Because the pseudo-transactions to the internal memory model are managed by QEMU, all we need to do is to send these pseudo-transactions to the bus model to pretend that the memory read/write operations are actually performed on the bus.

No matter what type the transaction belongs to, the read-/write commands of both types are only a hint, not a real operation applied on the hardware models. Thus, the pseudo-sockets mentioned above are not required to be paired with the target models except the data port for memory-mapped I/O access. Therefore, we preserve the primary template parameters of simple bus for the sockets which need to be paired and add new template parameters for the pseudo-initiator sockets which need not be paired with the target models. The new simple bus template is as given below.

```
1 template <int NR_OF_INITIATORS, int NR_OF_TARGETS, int
        NR_OF_IBUS, int NR_OF_DBUS>
2 class SimpleBusAT_QSC2 : public sc_core::sc_module
3 {
4 public:
5     typedef tlm::tlm_generic_payload transaction_type;
6     typedef tlm::tlm_phase phase_type;
```

978-1-4244-9021-9/11/$26.00 ©2011 IEEE

TABLE I. THE MAPPED INTERFACE AND UTILITIES OF DIFFERENT LEVEL MODELING.

Coding style	Interface		Utility	
			quantum keeper	payload event queue
Temporal Decoupling	`b_transport()`		Yes	–
Loosely-Timed	`b_transport()`		–	–
Approximately-Timed	initiator	`nb_transport_fw()`	–	Yes
	target	`nb_transport_bw()`		

```
7   typedef tlm::tlm_sync_enum sync_enum_type;
8   typedef tlm_utils::simple_target_socket_tagged<SimpleBusAT_QSC2>
        target_socket_type;
9   typedef tlm_utils::simple_initiator_socket_tagged<SimpleBusAT_QSC2
        > initiator_socket_type;
10 public:
11   target_socket_type target_socket[NR_OF_INITIATORS];
12   initiator_socket_type initiator_socket[NR_OF_TARGETS];
13   target_socket_type ibus_socket[NR_OF_IBUS];
14   target_socket_type dbus_socket[NR_OF_DBUS];
15   ...
16   uint32_t insn_cnt[NR_OF_IBUS];
17   uint32_t ld_cnt[NR_OF_DBUS];
18   uint32_t st_cnt[NR_OF_DBUS];
19   ...
20 }
```

The pseudo-target socket of the modified simple bus, `ibus_socket` and `dbus_socket`, has the same type of `target_socket` but is parameterized by the template parameters `NR_OF_IBUS` and `NR_OF_DBUS`. We use `ibus_socket` to receive the transactions on the instruction bus and `dbus_socket` to receive the transactions on the data bus. Also, the performance counters `insn_cnt`, `ld_cnt` and `st_cnt` are used to record, respectively, the number of instructions executed, the number of load and store operations performed. In essence, `ibus_socket` and `dbus_socket` both use the transport interface except that the performance counters are different.

Although the simple bus is modeled using AT coding style, the transport function `NBTransport_fw()` will actually call the LT transport function `BTransport()`. Finally, the virtual platform that integrates the IA-ISS and DMAC model, i.e., the `model.arm_n` and `model.dmac` instance, with the adapted simple bus, i.e., the `SimpleBusAT_QSC2<1,1,1,3>`. The model instantiation is as given below.

```
1  ...
2  soc_model model("model");
3  SimpleBusAT_QSC2<1,1,1,3> bus("bus");
4  ...
5  model.arm_busio->socket(bus.target_socket[0]);
6  model.arm_ibus->socket(bus.ibus_socket[0]);
7  model.arm_dbus->socket(bus.dbus_socket[0]);
8  model.dmacm1->socket(bus.dbus_socket[1]);
9  model.dmacm2->socket(bus.dbus_socket[2]);
10 bus.initiator_socket[0](model.busio_dmac->socket);
11 ...
```

The initiator sockets of IA-ISS and DMAC model need only call `b_transport()` the implementation of which resides in the target socket of the modified simple bus. Also notice that the only path that will transfer the transaction from the initiator model to the target model via the bus model is the following sequence: `model.arm_busio->socket` ⇌ `bus.target_socket[0]` ⇌ `bus.initiator_socket[0]` ⇌ `model.busio_dmac->socket`. The double arrows between the sockets indicate the direction of the forward and the backward interface used to transfer the transactions.

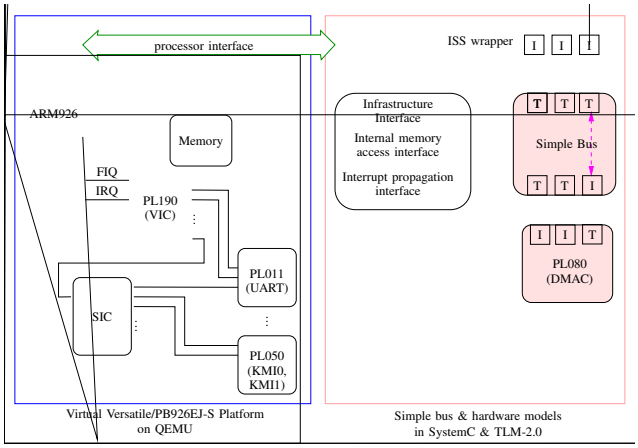


Figure 3. The block diagram of the virtual platform Versatile/PB926EJ-S of QEMU combined with the models written in SystemC and TLM-2.0.

TABLE II. NOTATIONS USED IN TABLE III

min	The best-case co-simulation time of 30 runs.
max	The worst-case co-simulation time of 30 runs.
μ	The mean of co-simulation time of 30 runs.
σ	The standard deviation of co-simulation time of 30 runs.
N_{TI}	The number of target instructions simulated.
N_{LD}	The number of load operations of the virtual processor.
N_{ST}	The number of store operations of the virtual processor.
N_{DMAR}	The number of read operations of DMAC.
N_{DMAW}	The number of write operations of DMAC.
N_{TX}	The total number of transactions.

IV. EXPERIMENTAL RESULTS

In this section, we turn our discussion to the experimental results of using the modified simple bus as the interconnect of the virtual Versatile/PB926EJ-S platform and as the interconnect of the IA-ISS and ARM PrimeCell PL080 DMAC [9] modeled in SystemC, the details of which are as shown in Fig. 3. Because PL080 has two master ports, M1 and M2, we use M1 to read and M2 to write data in turn.

Because the arbitration, the burst transfer, and the split transfer are not modeled in the simple bus, we only model the signal pins corresponding to the attributes of a transaction. In addition, the processor model is based on the ARM9 processor without cache. Because Linux kernel dose not provide the device driver for PrimeCell PL080 of the Versatile/PB926EJ-S platform, we also develop our own DMAC device driver as a char device for the purpose of testing, and its behavior can be controlled by an application program by calling the `ioctl()` function defined within the driver.

We provide two different metrics to measure the performance of the proposed virtual platform: (1) the simulation

TABLE III. SIMULATION TIME OF BOOTING UP THE LINUX KERNEL WITH DMA TEST BENCH USING TLM-2.0 SIMPLE BUS FOR 30 TIMES.

Statistics	Co-simulation time	N_{TI}	N_{LD}	N_{ST}	N_{DMAR}	N_{DMAW}	N_{TX}
min	40m50.756s	744,864,386.00	254,566,331.00	135,886,958.00	1,024,000.00	1,024,000.00	1,137,365,675.00
		(65.49%)	(22.38%)	(11.95%)	(0.09%)	(0.09%)	(100.00%)
max	44m20.429s	789,182,764.00	271,736,320.00	147,344,322.00	1,024,000.00	1,024,000.00	1,210,311,406.00
		(65.20%)	(22.45%)	(12.17%)	(0.08%)	(0.08%)	(100.00%)
μ	42m05.767s	759,358,111.80	260,203,215.90	139,598,175.97	1,024,000.00	1,024,000.00	1,161,207,503.67
		(65.39%)	(22.41%)	(12.02%)	(0.09%)	(0.09%)	(100.00%)
σ	00m49.962s	10,957,387.63	4,197,037.64	2,815,094.29	0.00	0.00	17,969,519.57

time it takes to boot up a full-fledged Linux kernel and to run the DMAC test bench, and (2) the statistics it can collect at instruction-accurate level while the system is being boot up. For all the experimental results given in this section, a 2.13GHz Intel Core 2 6420 processor machine with 3.2GB of memory is used as the host, and the target OS is built using the BuildRoot package [10]. The Linux distribution is Gentoo, and the kernel is Linux version 2.6.30. QEMU version 0.11.0-rc1, SystemC version 2.2.0 (including the reference simulator provided by OSCI), and TLM-2.0 (released on July 15, 2009) are all compiled by gcc version 4.3.2.

A. Time to Boot up Linux

In order to gather the statistics, the initial shell script is modified to enable the option of executing the DMAC test bench and rebooting the virtual machine automatically as soon as the booting sequence is completed. Furthermore, the pre-defined `no-reboot` option of QEMU will catch the reboot signal once the OS completed the DMAC test bench and started the reboot command to shut the QEMU down. Thus, the test bench can easily estimate the co-simulation time of QEMU and SystemC at the OS level.

Our test bench uses DMA to move data between memory allocated from the kernel space. The amount of data moved is 2,048,000 words, one half of which are read and the other half of which are write. Moreover, each word occupies 4 bytes. The simulation results of the simple bus is shown in Table III, and the notations used in Table III can be found in Table II.

As Table III show, the time taken to boot up a full-fledged Linux kernel is as shown in the column labeled "Co-simulation time." The column labeled "N_{TX}" gives the total number of target instructions executed and load and store operations performed. Because the number of read/write operations of the slave port of DMAC (PL080 in this case) has been counted as the load and store operations of the virtual processor, only the number of read/write operations of the master ports has to be counted. That is,

$$N_{\text{TX}} = N_{\text{TI}} + N_{\text{LD}} + N_{\text{ST}} + N_{\text{DMAR}} + N_{\text{DMAW}}.$$

The percentages given in parentheses are defined by

$$\frac{N_\alpha}{N_{\text{TX}}} \times 100\%$$

where the subscript α is either TI, LD, ST, DMAR, DMAW or TX.

V. CONCLUSION AND FUTURE WORK

This paper presents a QEMU and SystemC-based virtual platform that is capable of using TLM-2.0 interface and estimating its performance with an OS up and running. The IA-ISS is used as the processor model and is connected with DMAC model via the modified simple bus. In addition, using TLM-2.0 as the interface can provide the compatibility of other hardware models with the same interface. Moreover, the proposed virtual platform can provide statistics—such as the number of instructions executed, memory accessed, and DMAC read/write operations—that are valuable for the system designers at the early stage of ESL design flow. Although the co-simulation takes 44m20.429s to boot up a full-fledged kernel is still acceptable in the worst case, one of our goals is to speedup the simulation speed of TLM-2.0 interface in the future.

ACKNOWLEDGMENT

This work was supported in part by National Science Council, Taiwan, ROC, under Contract No. NSC99-2221-E-110-052.

Rnloa003edg013 agreem65(=tlmd( )TjETq10(a.55esi27 30e.8