# A Novel Technique for Making QEMU an Instruction Set Simulator for Co-simulation with SystemC

Tse-Chen Yeh, Zin-Yuan Lin, and Ming-Chao Chiang

***Abstract*—This paper presents a novel technique for converting QEMU from a virtual machine into an instruction-accurate instruction set simulator (IA-ISS) and using it as the processor model of a QEMU and SystemC-based virtual platform. The proposed framework can not only simulate arbitrary hardware modeled in SystemC, but it can also be used to evaluate the performance of the target system for SoC development. Our experimental results show that the built-in vector interrupt controller of QEMU modeled in C can be easily replaced by one modeled in SystemC for demonstrating the waveform of AMBA on-chip-bus model connected with the adapted IA-ISS. Moreover, the instruction-accurate statistics can be gathered while co-simulating with a full-fledged Linux kernel. Our experimental results further show that with every instruction executed and every memory accessed since power on traced, the hardware/software co-simulation takes no more than 16 minutes in booting up the Linux kernel, even in the worst case.**

***Index Terms*—QEMU, SystemC, ISS, platform-based design, SoC.**

## I. Introduction

DUE to the system complexity, the cost consideration, and the time-to-market pressure, most of the system-on-chip (SoC) designs tend to adopt the platform-based methodologies for enabling the system software development and the system performance evaluation at the early stage of electronic system level (ESL) design flow [1]. Proposed in 2007, the focus of the QEMU-SystemC framework was on system software and device driver development [2]. Although it is suitable for developing hardware accelerators, it is insufficient for evaluating the system performance because no information about the processor is provided. Several works on enhancing the QEMU-SystemC wrapper are proposed, such as TLM interface appended [3] and the combination with CoWare's Platform Architect [4], [5], [6]; however, none of them are capable of evaluating the performance of a target system from the system perspective. As a result, system designers have no idea about the number of instructions and the number of load/store operations executed when running an operating system (OS) on the target system that is currently under development.

Instead of looking for an applicable instruction set simulator (ISS) to build a virtual platform from scratch, we decide to go the other way around by converting QEMU as a virtual machine to an instruction-accurate ISS (IA-ISS) that can be used as a processor model of existing virtual platforms. The I/O interface exported by QEMU-SystemC works seamlessly with the ISS we describe herein. The consequence is that the virtual platform constructed in this way is more suitable for performance evaluation and design space exploration than QEMU-SystemC, especially from the system perspective.

Manuscript received December 8, 2010; revised January 8, 2011. This work was supported in part by the National Science Council, Taiwan, ROC, under Contracts NSC98-2221-E-110-049 and NSC99-2221-E-110-052.

The authors are with the Department of Computer Science and Engineering, National Sun Yat-sen University, Kaohsiung 80424, Taiwan, ROC. (e-mail: sdgp03@ms18.hinet.net, m983040031@student.nsysu.edu.tw, mcchiang@cse.nsysu.edu.tw).

## II. Related Work

In this section, we begin with a brief introduction to SystemC and QEMU. Then, we turn to the QEMU-SystemC framework and its variants for the SoC development. Finally, we briefly discuss the pros and cons of the dynamic binary translation (DBT) technique used by QEMU, which eventually make the conversion of QEMU from a system emulator to an ISS much more difficult than anticipated.

### A. SystemC

SystemC is an ANSI standard C++ class library developed by Open SystemC Initiative (OSCI) [7] in 1999 and approved as IEEE standard in 2005 [8]. Although relatively new, SystemC has become one of the most popular modeling languages in the ESL design flow [9]. Because SystemC can simulate concurrency, events, and signals of a hardware, the abstraction of the hardware model can be achieved up to the transaction level without the need of considering the signal level details, thus making it a perfect foundation for a virtual platform.

### B. QEMU

QEMU is an open source system emulator [10] capable of emulating several target CPUs on several hosts. Moreover, quite a few OSs have been ported to the virtual platforms supported by QEMU. The functional diagram of the virtual platforms supported by QEMU is as shown in Fig. 1. Basically, all the virtual platforms are composed of the processor model, software MMU, internal memory model, memory-mapped I/O models, and interrupt cascading. The software MMU is used to translate the virtual address of the target processor into individual memory-mapped address of internal memory and I/O models. The interrupt cascading is used to chain the interrupt lines from the downstream components to the upstream components.

### C. QEMU-SystemC

QEMU-SystemC [2] is an open source software/hardware emulation framework for the SoC development. It allows

ISBN: 978-988-18210-3-4
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

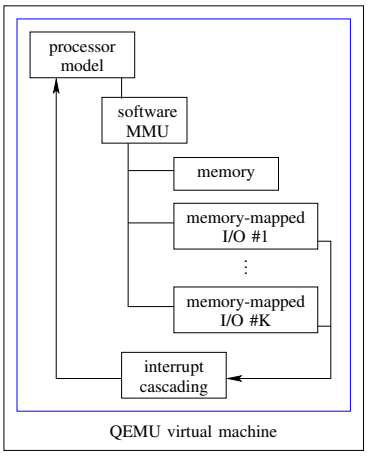



Fig. 1. Functional diagram of the virtual platforms supported by QEMU [10].

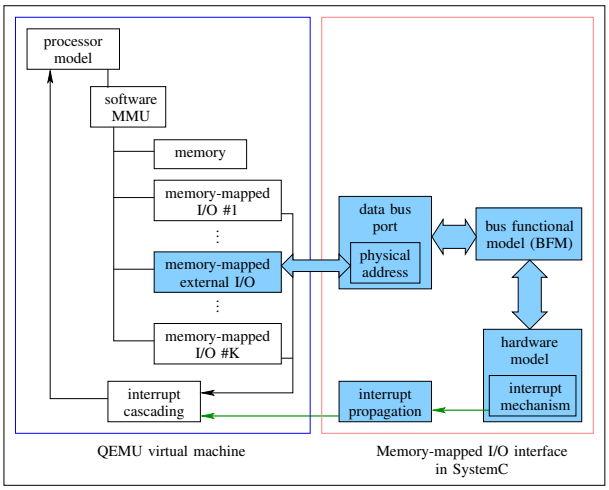



Fig. 2. Functional diagram of QEMU-SystemC [2]. The memory-mapped external I/O interface corresponds to the AMBA interface in the original paper from the implementation perspective.

hardware models to be inserted into specific memory-mapped addresses of QEMU and communicates by means of the "memory-mapped external I/O interface" as shown in Fig. 2. Although the QEMU-SystemC framework can be used to trace the access of attached hardware models, no information about the processor is available for the virtual platform. For instance, the instructions executed, the memory accessed, and so on, which can be valuable to the system designers, are unfortunately not provided.

### D. QEMU-SystemC with CoWare

Another framework [4], [5], [6] combines the enhanced QEMU-SystemC wrapper with CoWare's Platform Architect [11]. The QEMU-SystemC wrapper communicates with the CoWare-SystemC wrapper by using the inter-process communication (IPC) socket interface. This framework utilizes the bus models provided by off-the-shelf Model Library [12], which supports lots of capabilities for profiling and analysis. However, no details whatsoever about the CoWare-SystemC wrapper they proposed, including the capability to estimate the performance of the processor model, are provided.

### E. Dynamic Binary Translation Used by QEMU

As shown in Fig. 3, the DBT used by QEMU for translating the target code to the host code is basically composed of two steps [10]. The details are as given below.

1) The first step, which is composed of three sub-steps, is to generate the dynamic code generator (DCG) off-line, as follows:
   a) The first sub-step is to split each of the target instructions into a sequence of *micro-operations*. Each micro-operation is then *hand-coded* as a tiny function (TF) in C.
   b) The second sub-step is to have the set of TFs compiled by gcc [13] into the host code in an object file in the host object file format such as ELF on Linux.
   c) The third sub-step is to launch tiny code generator (`TCG`)—which takes as input the object file containing all the micro-operations—to generate the dynamic code generator (DCG).
2) The second step, which consists of two sub-steps, is to translate the target code to the host code at run-time, as follows:
   a) The first sub-step is to translate each of the target instructions into a sequence of indices each of which uniquely identifies a TF.
   b) The second sub-step is for the DCG to copy the host code corresponding to each TF the index of which is given in the first sub-step to translation block (TB), in the order as given in the first sub-step and with constant parameters patched.

In short, the DBT used by QEMU can be divided into two steps the first of which is the *preparation* step while the second of which is the *translation* step. The first step is responsible for generating the DCG off-line. The second step is responsible for actually translating each of the target instructions into the host code—by first translating the target instructions into a sequence of indices to TFs and then from the sequence of indices into the host code. Then, the host code in the translation block cache (TBC) can be executed directly.

## III. The Proposed Method

In this section, we will first present the technique for converting QEMU from a virtual machine into an IA-ISS, followed by the technique to make such an IA-ISS a processor model of a virtual platform. Then, we will discuss how QEMU and SystemC are integrated.

### A. From Virtual Machine to IA-ISS

To convert QEMU from a virtual machine to an IA-ISS, we have to make use of the so-called helper functions provided by TCG instead of the hand-coded tiny functions mentioned in Section II-E. The purpose of the helper functions of TCG is to wrap up whatever functions to be called such as the library functions provided by QEMU to ensure that they comply with the coding rules imposed by the TCG of QEMU version 0.10.$x$ or later.

Instead of giving the low-level details, we will focus on the high-level view of what has to be done. To simplify our

ISBN: 978-988-18210-3-4
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

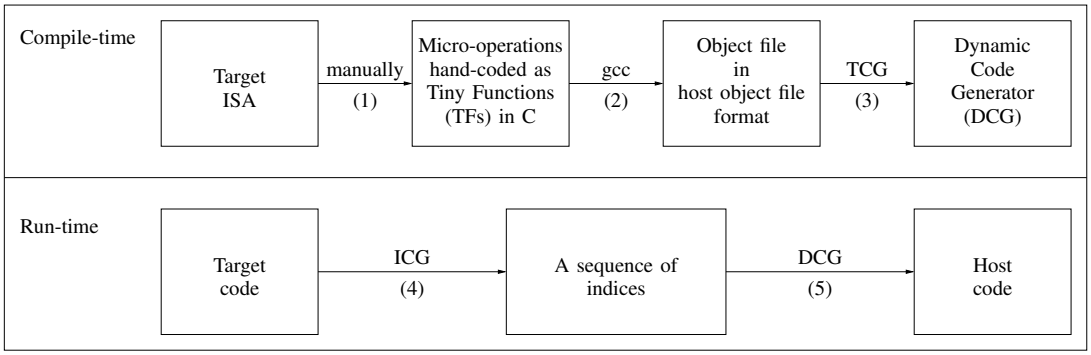



Fig. 3. Steps taken by QEMU to dynamically translate the target code to the host code. The first three steps are done at compile-time while the last two steps are invoked at run-time.

discussion that follows, let us assume that $f$ is the name of the helper function to be defined, $t_r$ the type of the return value, and $t_i$ the type of parameter $i$.

1) For each helper function $f$ to be defined, the first thing to do is to use the macro

   `DEF_HELPER_`$n(f, t_r, t_1, \ldots, t_n)$[1]

   to generate three pieces of code: (1) the prototype of the helper function `helper_`$f$, (2) the 'op' helper function `gen_helper_`$f$ to be called by DBT to generate the host code to call the helper function, and (3) the code to register the helper function at run-time for the purpose of debugging. For instance, the macro

```
1  DEF_HELPER_2(fetch_insn, void, i32, i32)
```

   will generate the following code.

```
1  void helper_fetch_insn(uint32_t, uint32_t);
2
3  static inline void
4  gen_helper_fetch_insn(TCGv_i32 arg1, TCGv_i32 arg2)
5  {
6      TCGArg args[2];
7      int sizemask;
8      sizemask = 0;
9      args[1 - 1] = GET_TCGV_I32(arg1);
10     sizemask |= 0 << 1;
11     args[2 - 1] = GET_TCGV_I32(arg2);
12     sizemask |= 0 << 2;
13     tcg_gen_helperN(helper_fetch_insn, 0, sizemask,
           TCG_CALL_DUMMY_ARG, 2, args);
14 }
15
16 tcg_register_helper(helper_fetch_insn, "fetch_insn");
```

   The prototype of the helper function is given in line 1. The 'op' helper function is defined in lines 3–14. The code to register the helper function for the purpose of debugging is invoked in line 16.

2) The second thing is to define the helper function `helper_`$f$ declared above—by wrapping up whatever to be executed inside the helper function. The helper function will get called by the host code generated by the 'op' helper function defined above and executed together with the host code of each target instruction, as shown in Fig. 6.

3) The third thing is to use the `tcg_const_i32` function or its variants to save values that vary from instruction to instruction at the time of DBT away so that they can be restored later on at run-time. Good examples are instructions executed and their operands.

[1]where $n >= 0$ is the number of input parameters, with $n = 0$ indicating that there is no input parameter.

Once we have all the helper functions and the 'op' helper functions for extracting the information SystemC needs defined, all we have to do is to find the right place to insert each 'op' helper function so that it will get called at the time of DBT to generate the host code to call the helper function associated with it. As such, the helper functions will be executed along with the target instructions. Since the key information an IA-ISS needs to provide consists of the address and data of target instructions fetched and memory accessed, we will discuss in detail how they are extracted below.

1) Target instruction fetch stage: All we have to do here is to insert an 'op' helper function—the parameters of which are the address of the target instruction and the target instruction itself—right after the place where the target instruction is being fetched for DBT. The 'op' helper function so inserted will get called at the time of DBT to generate the host code—and place it right before each target instruction—to call the associated helper function and pass it the parameters above so that the helper function can send the address of the target instruction and the target instruction itself to SystemC, as shown in Fig. 6 (cf. Fig. 5). Note that the notations used in Figs. 5 and 6 are summarized in Table I. This stage can be imagined as an "Instruction Fetch eXtractoR" (IFXR) that can assist in extracting the information hidden in the processor model shown in Fig. 4.

2) Memory access stage: All we have to do here is to define two helper functions: one to extract the address and one to extract the data. For memory load, the 'op' helper functions for extracting the address and data will be inserted, respectively, right before and right after the memory load function. For memory store, the 'op' helper functions for extracting the address and data will be inserted right before the memory store function. After that, the 'op' helper functions will be called by the DBT to generate the host code to call the corresponding helper functions, and the results are as shown in Fig. 6(b) and (c). This stage can be viewed as a "Memory Access eXtractoR" (MAXR) that can assist in extracting the access transactions between the software MMU and the internal memory-mapped models shown in Fig. 4.

Since the sole purpose of the helper functions is to extract the address and data of the target instructions executed by

ISBN: 978-988-18210-3-4
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

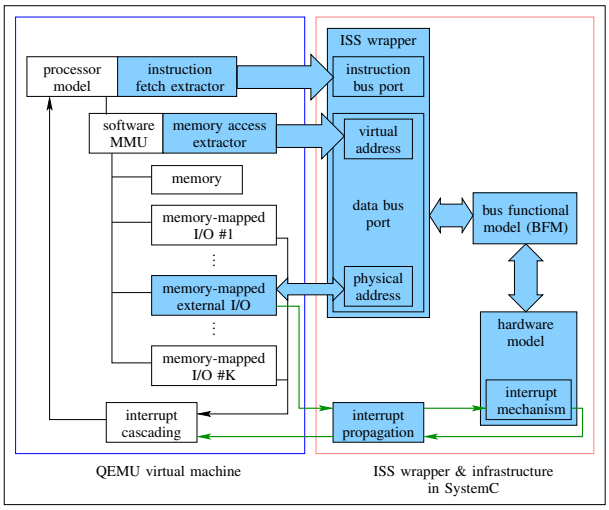



Fig. 4. Functional diagram of an IA-ISS based on QEMU.

TABLE I
NOTATIONS USED IN FIGS. 5 AND 6

| | |
|---|---|
| TB | Translation block. |
| TI | Target instruction. |
| TF | Tiny function. |
| IXC | Information extraction code. |
| TB′ | TB with IXC. |
| $IXC_a^{ml}$ | Memory load address IXC. |
| $IXC_d^{ml}$ | Memory load data IXC. |
| $TF^{ml}$ | TF for memory load, i.e., with $IXC_a^{ml}$ and $IXC_d^{ml}$. |
| $IXC^{ms}$ | Memory store IXC. |
| $TF^{ms}$ | TF for memory store, i.e., with $IXC^{ms}$. |
| TBC | TB cache, which is composed of one or more TBs, chained together. |

QEMU instead of modifying their values, the behavior of executing the target instructions is guaranteed to be the same as if the helper functions were not there, except that it takes longer to execute now because it consists of not only the "original" host code but also the host code for extracting the information SystemC needs. Moreover, the transactions extracted by IFXR can be imagined as the transactions on the instruction bus while the transactions extracted by MAXR can be viewed as the transactions on the data bus.

### B. Infrastructures for Virtual Platform Construction

To fulfill the requirements of being a system emulator, QEMU provides an I/O interface for porting the target processor into different virtual platforms. Although undocumented, most of the existing virtual platforms have been modeled and constructed based on this I/O interface, which can be divided into two categories: PCI and memory-mapped I/O. They are suitable for different target processors. For QEMU-SystemC [2] and the framework we presented herein, the I/O communications between hardware models of QEMU and hardware models in SystemC are implemented based on this I/O interface.

The built-in interrupt mechanism of QEMU requires that each interrupt handler be registered by calling the `qemu_allocate_irqs()` function, the return value of which is a pointer used by the downstream peripherals to find their upstream components when they need to announce an interrupt. The registered interrupt handler is usually used

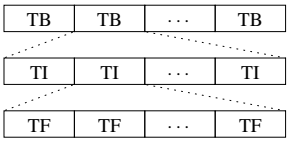



Fig. 5. TBC before insertion of IXC (cf. Fig. 6). Note that the lower layer is an enlargement of its immediate upper layer. In other words, the upper layer is composed of an unknown number of the lower layer chained together. In short, TBC is composed of an unknown number of TFs.

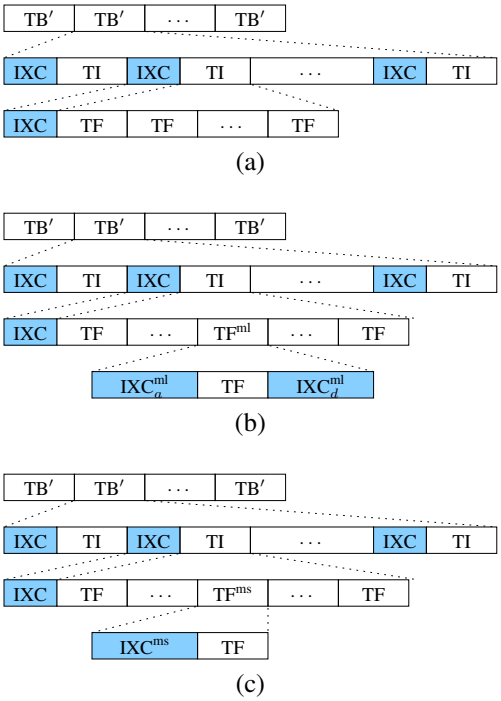



Fig. 6. TBC after insertion of IXC (cf. Fig. 5). Note that the lower layer is an enlargement of its immediate upper layer. (a) Non-load and non-store instructions, (b) Memory load instructions, and (c) Memory store instructions.

to receive the interrupt from the downstream components. After the input interrupt signals are processed, the results can be sent to the upstream components by using the function `qemu_set_irq()`. All the virtual hardware devices in QEMU use this mechanism to cascade all the interrupt controllers together to form an interrupt hierarchy. In order to maintain the built-in interrupt hierarchy after models of QEMU are replaced by models written in SystemC, all we need to modify is the virtual hardware initialization and the interrupt handler provided for the downstream peripherals. The overall functional block of the virtual platform with the adapted IA-ISS is as shown in Fig. 4. The interrupt propagation is a little different from the one QEMU-SystemC provides because the hardware model used in our experiments requires both the sending and receiving directions to be supported.

### C. Integration of QEMU and SystemC

We implement QEMU and SystemC as two threads since context switches between threads are generally much faster than between processes. Moreover, the shared memory mechanism is adopted and used as a FIFO between QEMU and SystemC as shown in Fig. 7. In other words, by design, the

ISBN: 978-988-18210-3-4
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

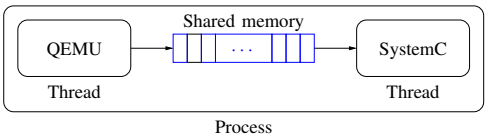



Fig. 7. Inter-process communication (IPC) mechanism used by the framework we presented in the paper.

communication between QEMU and SystemC is unidirectional so that the relative order of the instructions executed, the memory accessed, and the I/O write operations performed is retained. Besides, the synchronization of the I/O read operations can be achieved by having QEMU call the I/O read function and then block until SystemC returns.

## IV. Experimental Results

In this section, we demonstrate the virtual platform with the proposed IA-ISS as the processor model in terms of three different measures: (1) co-simulation time it takes to boot up a Linux kernel, (2) statistics that can be collected while the system is up and running, and (3) waveform that can help system designers debug activities between the processor and all the hardware models. For all the experimental results given in this section, a 2.00 GHz Intel Core 2 Duo T7200 processor machine with 1.5GB of memory is used as the host, and the target OS is built using the BuildRoot package [14]. The Linux distribution is Fedora 11, and the kernel is Linux version 2.6.29.4-167. QEMU version 0.10.5 and SystemC version 2.2.0 are all compiled by gcc version 4.4.0.

### A. Virtual Platform for Co-Simulation

In order to verify the system functionality, the proposed virtual platform is demonstrated by replacing the PrimeCell PL190 vector interrupt controller (VIC) [15] of the virtual platform provided by QEMU with a PL190 VIC written in SystemC, as shown in Fig. 8. The interrupt hierarchy of the Versatile/PB926EJ-S development board [16] is composed of the PL190 VIC, the secondary interrupt controller (SIC), the PrimeCell PL011 UART controller, and the PL050 keyboard mouse interface (KMI) controller. Since PL190 VIC is the one directly interacted with ARM926EJ-S, the correctness of its implementation in SystemC can be easily verified. In other words, any bugs found in PL190 VIC written in SystemC will eventually crash the OS in question sooner or later. Thereby, a few runs of ARM Linux on the virtual Versatile/PB926EJ-S platform of QEMU will essentially give us a hint regarding the correctness of the PL190 VIC in SystemC.

### B. Co-Simulation Time and Statistics

In order to gather the statistics, the initial shell script was modified to reboot the kernel automatically as soon as the booting sequence is completed. The pre-defined `no-reboot` option of QEMU will catch the reboot signal once the OS executes the reboot command and then shuts the QEMU down. Thus, the test bench can easily estimate the co-simulation time of QEMU and SystemC at the OS level.

The co-simulation times shown in the column labeled "Co-simulation time" of Table II are collected using the Linux's

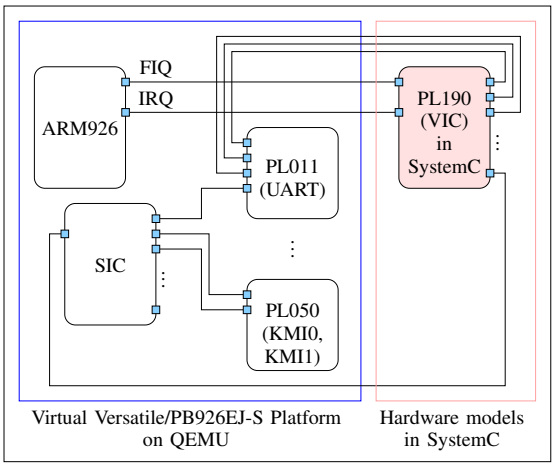



Fig. 8. Block diagram of virtual Versatile/PB926EJ-S platform on QEMU with the PL190 VIC model (in C) replaced by a hardware model written in SystemC.

`time` command. From the perspective of the hardware designer, it is probably much faster than just acceptable to co-simulate a full-fledged operating system in less than 15 minutes on average and less than 16 minutes in the worst case, especially in the early stage of SoC development. Moreover, the notations used in Table II are listed in Table III.

As shown in Table II, the column labeled "$N_{\mathrm{TI}}$" shows the number of target instructions actually executed by the virtual ARM processor. The columns labeled "$N_{\mathrm{LD}}$" and "$N_{\mathrm{ST}}$" present, respectively, the number of load and store operations of the virtual processor including the memory-mapped I/O.

The column labeled "$N_{\mathrm{TI+LD+ST}}$" gives the total number of target instructions executed and load and store operations performed because the number of the read/write operations of the VIC (PL190 in this case) have been counted as the load and store operations of the virtual processor. That is, $N_{\mathrm{TI+LD+ST}} = N_{\mathrm{TI}} + N_{\mathrm{LD}} + N_{\mathrm{ST}}$. The columns labeled "$N_{\mathrm{RD}}$" and "$N_{\mathrm{WT}}$" give an idea about the number of read/write transactions between the virtual processor and the VIC. They are given to show that the proposed virtual platform can provide information as detail as the number of the read/write operations of the VIC. Note that all the numbers given are, as the names of the rows suggest, the $\min$, $\max$, and average of booting up the ARM Linux and shutting it down immediately on the proposed framework for 30 times.

The percentages given in parentheses are computed as

$$\frac{N_{\alpha}}{N_{\mathrm{TI+LD+ST}}} \times 100\%$$

where the subscript $\alpha$ is either TI, LD, ST, TI + LD + ST, RD, or WT. The others are computed similarly. They are shown to give an idea about how many percent of all the target instructions executed and all the load/store operations performed are target instructions, how many of them are load and store operations, and so on.

### C. Waveform of AMBA On-Chip Bus

The primary differences between QEMU-SystemC and the framework we described herein are as shown by the waveforms of the address and data signals by using exactly

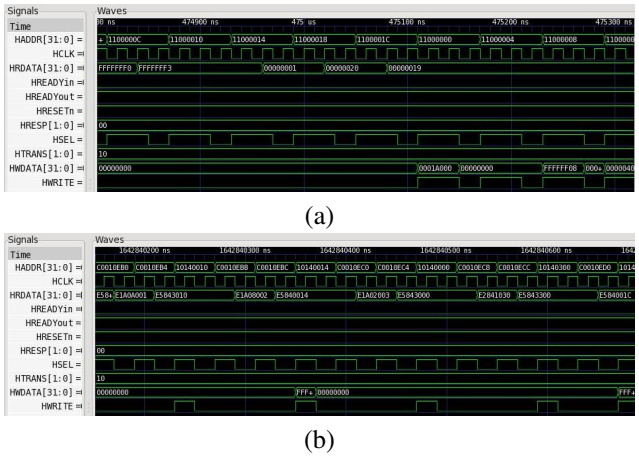
ISBN: 978-988-18210-3-4
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

TABLE II
EXPERIMENTAL RESULTS OF BOOTING UP THE LINUX KERNEL ON THE PROPOSED VIRTUAL PLATFORM FOR 30 TIMES.

| Statistics | Co-simulation time | $N_{TI}$ | $N_{LD}$ | $N_{ST}$ | $N_{TI+LD+ST}$ | $N_{RD}$ | $N_{WT}$ |
|---|---|---|---|---|---|---|---|
| min | 14m01.994s | 658,473,617.00 | 231,778,725.00 | 78,616,811.00 | 968,869,153.00 | 132,556.00 | 198,556.00 |
| | | (67.96%) | (23.92%) | ( 8.11%) | (100.00%) | (0.01%) | (0.02%) |
| max | 15m37.327s | 701,925,646.00 | 244,722,965.00 | 86,537,514.00 | 1,033,186,125.00 | 150,312.00 | 225,370.00 |
| | | (67.94%) | (23.69%) | ( 8.38%) | (100.00%) | (0.01%) | (0.02%) |
| $\mu$ | 14m49.317s | 672,447,292.77 | 236,356,464.13 | 81,375,519.70 | 990,179,276.60 | 141,016.27 | 211,326.10 |
| | | (67.91%) | (23.87%) | ( 8.22%) | (100.00%) | (0.01%) | (0.02%) |
| $\sigma$ | 00m22.461s | 13,997,515.54 | 3,911,802.32 | 2,405,392.30 | 20,314,710.16 | 4,264.56 | 6,421.29 |

TABLE III
NOTATIONS USED IN TABLE II

| | |
|---|---|
| min | The best-case co-simulation time of 30 runs. |
| max | The worst-case co-simulation time of 30 runs. |
| $\mu$ | The mean of co-simulation time of 30 runs. |
| $\sigma$ | 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. |
| $N_{RD}$ | The number of times the virtual processor reads data from the VIC (PL190). |
| $N_{WT}$ | The number of times the virtual processor writes data to the VIC (PL190). |




(a)

(b)

Fig. 9. Snapshot of the waveform of AMBA on-chip bus. (a) Snapshot of QEMU-SystemC. (b) Snapshot of the proposed framework.

the same AMBA bus protocol. QEMU-SystemC can only provide the signal transitions of I/O in Fig. 9(a) whereas the framework we described herein can simulate not only the signal transitions of I/O but also all the instructions executed and all the memory accessed since power on as shown in Fig. 9(b). Moreover, the HADDR, HRDATA, and HWDATA reveal that the operations alternate between the interconnect, processor, and virtual hardware modeled in SystemC. The addresses in the range of 0x1014_0000 to 0x1014_0400 are accessed to the memory-mapped I/O of PL190 VIC modeled in SystemC. Note that the waveforms given in Fig. 9 is shown by another open source tool called GTKWave [17], which reads standard Verilog VCD/EVCD files and allows their contents to be viewed.

## V. CONCLUSION

This paper presents a novel technique for converting QEMU from a virtual machine into an IA-ISS for a QEMU and SystemC-based virtual platform that overcomes the limitations of QEMU-SystemC. Not only can the proposed virtual platform be used to evaluate the performance of a target system, but it can also be used for design space exploration at the instruction level. The waveform output reveals the potential for modeling the bus functional model on the proposed virtual platform. Moreover, our experimental results show that even in the worst case, the proposed virtual platform takes no more than 16 minutes to boot up a full-fledged Linux kernel, indicating that the proposed virtual platform provides a fast solution for the hardware designer, especially in the early stage of the SoC development.

## ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers for their valuable comments and suggestions on the paper.

## REFERENCES

[1] B. Bailey, G. Martin, and A. Piziali, *ESL Design and Verification*. Morgan Kaufmann Publishers, 2007.
[2] M. Montón, A. Portero, M. Moreno, B. Martínez, and J. Carrabina, "Mixed SW/SystemC SoC emulation framework," in *Proceedings of IEEE International Symposium on Industrial Electronics*, Jun. 2007, pp. 2338–2341.
[3] M. Montón, J. Carrabina, and M. Burton, "Mixed simulation kernels for high performance virtual platforms," in *Proceedings of Forum on Specification and Design Languages*, Sep. 2009, pp. 1–6.
[4] C.-C. Wang, R.-P. Wong, J.-W. Lin, and C.-H. Chen, "System-level development and verification framework for high-performance system accelerator," in *Proceedings of International Symposium on VLSI Design, Automation and Test*, Apr. 2009, pp. 359–362.
[5] J.-W. Lin, C.-C. Wang, C.-Y. Chang, C.-H. Chen, K.-J. Lee, Y.-H. Chu, J.-C. Yeh, and Y.-C. Hsiao, "Full system simulation and verification framework," in *Proceedings of Fifth International Conference on Information Assurance and Security*, Aug. 2009, pp. 165–168.
[6] S.-T. Shen, S.-Y. Lee, and C.-H. Chen, "Full system simulation with qemu: An approach to multi-view 3d gpu design," in *Proceedings of 2010 IEEE International Symposium on Circuits and Systems*, May 2010, pp. 3877–3880.
[7] OSCI. Open SystemC Initiative. http://www.systemc.org/.
[8] IEEE Computer Society. (2005) IEEE standard SystemC language reference manual. Design Automation Standards Committee. http://standards.ieee.org/getieee/1666/download/1666-2005.pdf.
[9] M. Creamer. Nine reasons to adopt SystemC ESL design. http://www.eetimes.com/showArticle.jhtml?articleID=47212187.
[10] F. Bellard, "QEMU, a fast and portable dynamic translator," in *Proceedings of USENIX Annual Technical Conference*, Jun. 2005, pp. 41–46.
[11] Synopsys Inc. Platform architect. http://www.synopsys.com/Tools/SLD/VirtualPrototyping/Pages/PlatformArch%itect.aspx.
[12] ——. Platform architect models. http://www.synopsys.com/Tools/SLD/VirtualPrototyping/SLLibraries/Pages/%PlatformArchitect.aspx.
[13] R. M. Stallman. The GNU Compiler Collection. http://gcc.gnu.org/.
[14] P. Korsgaard. BuildRoot. http://buildroot.uclibc.org/.
[15] ARM Ltd. (2000) ARM PrimeCell Vectored Interrupt Controller (PL190) Technical Reference Manual. http://infocenter.arm.com/help/index.jsp.
[16] DesignA Electronics. RealView Versatile/PB926EJ-S. http://www.bluewatersys.com/development/doc/realview/versatile/pb.php.
[17] T. Bybell. GTKWave. http://gtkwave.sourceforge.net/.