A Fast Cycle-Accurate Instruction Set Simulator Based on QEMU and SystemC for SoC Development

Tse-Chen Yeh [#1], Guo-Fu Tseng [#2], Ming-Chao Chiang [#3]

[#] *Department of Computer Science and Engineering, National Sun Yat-sen University*
70 Lienhai Road, Kaohsiung 80424, Taiwan, R.O.C.

[1] sdgp03@ms18.hinet.net
[2] cooldavid@cooldavid.org
[3] mcchiang@cse.nsysu.edu.tw

***Abstract*—This paper presents a fast cycle-accurate instruction set simulator (CA-ISS) based on QEMU and SystemC. The CA-ISS can be used for design space exploration and as the processor core for virtual platform construction at the cycle-accurate level. Even though most state-of-the-art commercial tools try to provide all the levels of details to satisfy the different requirements of the software designer, the hardware designer, or even the system architect, the hardware/software co-simulation speed is dramatically slow when co-simulating the hardware models at the register-transfer level with a full-fledged operating system. In this paper, we show that the combination of QEMU and SystemC can make the co-simulation at the cycle-accurate level extremely fast, even with a full-fledged operating system up and running. Our experimental results indicate that with every instruction executed and every memory accessed since power-on traced at the cycle-accurate level, it takes less than 17 minutes on average to boot up a full-fledged Linux kernel, even on a laptop.**

I. Introduction

To cope with the exponential complexity of SoC development nowadays and in the future, Electronic System Level (ESL) design flow [1] was established to capture design intent at a higher level of abstraction than the traditional RTL design flow. It is, however, not until the emergence of SystemC that the hardware and software are bumped together and the co-simulations between the hardware and software are made seamlessly. Defined as an ANSI standard C++ class library for system and hardware design, SystemC is certainly one of the most popular system level description languages (SLDLs) nowadays because it provides a common language for both the hardware and software designers. Even though many electronic design automation (EDA) companies deliver platform-based hardware/software co-simulation environments, which support SystemC, the hardware/software co-simulation speed is dramatically slow when co-simulating the hardware models at the register-transfer level with a full-fledged operating system up and running. In addition, several other tools use ISS of a specific processor. For instance, SPACE and MPARM support ARM ISS simulation [2], [3]. Platune supports MIPS architecture [4]. In 2007, QEMU-SystemC is proposed as a software/hardware SoC emulation framework [5] to leverage the strengths of QEMU and SystemC. The original intent of QEMU-SystemC is to facilitate the development of software and device drivers for whatever Operating System (OS) that happens to be running on QEMU without spending too much effort on modifying the virtual platform itself.

Although software development can benefit from such a combination, several constraints imposed by QEMU-SystemC make the design space exploration extremely difficult when more accurate levels are required. First, the transactions of the on-chip bus model do not carry sufficient processor information such as instructions executed and memory accessed. Second, it is incapable of providing simulation between the processor and virtual devices at the cycle-accurate level. To overcome these constraints and to make it a much more flexible framework for design space exploration and virtual platform construction, we describe in this paper a framework to overcome these issues—by making QEMU a fast cycle-accurate ISS (CA-ISS).

II. Related Work

In this section, we begin with a brief introduction to SystemC and then discuss ISS for system emulation. In order to ensure that the framework described herein is capable of running an OS, we consider only ISSs that are able to emulate a system. We also present several techniques used in making QEMU a system emulator. Finally, we will briefly discuss the QEMU-SystemC framework.

A. SystemC

SystemC is an ANSI standard C++ class library developed by Open SystemC Initiative (OSCI) [6] in 1999 and approved as IEEE standard in 2005 [7]. Although SystemC is a relatively new standard, it has become one of the most popular modeling languages in the ESL design flow. Because SystemC can simulate concurrency, events, and signals of a hardware [8], a higher level of abstraction of the hardware model such as abstraction at the transaction level can be easily

978-1-4244-5794-6/10/$26.00 ©2010 IEEE

achieved. As a result, the co-simulation speed can be highly enhanced. Moreover, the instruction- and cycle-accurate ISS, as we described herein, requires these features of SystemC to communicate directly with the hardware models.

B. Internals of QEMU

QEMU [9] is an open source system emulator, which can emulate several target CPUs on several hosts. Moreover, quite a few OSs have been ported to the virtual platforms supported by QEMU. There are eventually several reasons why we choose QEMU to be the ISS. First comes the simulation speed of QEMU that is fast even with an OS running on it. Then comes the fact that QEMU supports more processors than other system emulators. Finally, it comes the QEMU-SystemC framework that proved the possibility of integrating QEMU and SystemC.

1) Target Instruction Simulation in QEMU: A highly simplified view of the CPU main loop of QEMU is as given in Fig. 1 assuming that the virtual machine is just powered on or reset. In that case, the interrupt signals of the virtual machine are all disabled, and the Translation Block Cache (TBC), which is composed of a collection of the so-called Translation Blocks (TBs), is empty. Then, as Fig. 1 shows, the very first thing the CPU main loop of QEMU does is to process all the pending interrupts, if any, by setting up calls to the corresponding interrupt service routines (ISRs) so that they will get translated by the so-called Dynamic Binary Translation (DBT) to the host code. Then, it will start executing whatever host code in the TBC. However, if it is a TBC miss, then the DBT will be launched. After that, the CPU main loop will be re-entered, and the execution of the host code will be re-started. This time, it will be a TBC hit because some of the host code just get dynamically translated into the TBC. The CPU main loop will loop forever until the virtual CPU aborts, which will in turn halt the processor.

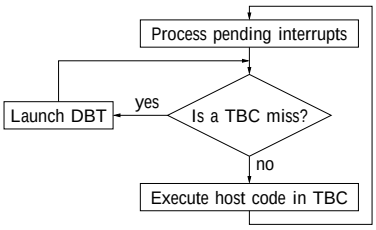


Fig. 1. A simplified view of the CPU main loop of QEMU

2) Hardware Emulation in QEMU: In order to simulate the behavior of a processor, we need to take into account the I/O. The way the load and store instructions of the target CPU access the memory depends on how the virtual address of the target OS is mapped to the virtual address of the host processor. As for the I/O, QEMU [10] has defined a set of callback functions in C to act as the I/O interface in the system mode, which can be used to model the virtual hardware device for the virtual platform of QEMU. Most of the hardware interrupt sources will be connected to the virtual interrupt controller modeled by the I/O interface of QEMU. Then, the virtual interrupt controller will ultimately be connected to the virtual CPU, which will in turn call a specific function asynchronously to inform the CPU main loop of QEMU that an interrupt is pending.

C. QEMU-SystemC

QEMU-SystemC [5] is an open source software/hardware 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 as shown in Fig. 2.

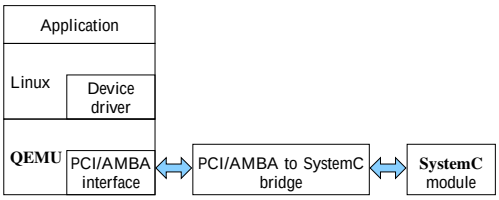


Fig. 2. The block diagram of QEMU-SystemC [5]. The functional descriptions of PCI/AMBA interface and PCI/AMBA to SystemC bridge in the block diagram are different from those in the original paper, but they are identical from the implementation perspective.

Although the waveform of the AMBA on-bus chip of the QEMU-SystemC framework can be used to trace the I/O, no information about the processor is made available for design space exploration of the system. For instance, the instructions executed, the addresses of the memory accessed, and so on, which can be valuable to the system designers, are unfortunately not provided. Moreover, QEMU-SystemC does not provide the cycle-accurate information that can be valuable to the hardware designers.

III. The Proposed Method

The original idea behind QEMU is to provide a virtual machine that can do whatever a physical machine can do. In other words, the virtual machine should be able to run a full-fledged OS and all the applications that can be run on the OS and run them fast. Therefore, most of the design considerations of QEMU are on enhancing the performance of the QEMU itself, especially when emulating an OS is considered. As such, several tricky techniques are used. However, such a design choice also increases the difficulty on making QEMU a cycle-accurate ISS we propose herein for design space exploration on the SoC development.

A. From Fast System Emulator to Fast Cycle-Accurate ISS

To make QEMU a CA-ISS, we have to send all the information the co-simulator needs such as the instructions executed and the data accessed including their addresses from QEMU to SystemC. Conceptually, this can be easily done by embedding instructions directly into the target code to be executed while they are being translated to the host code. However, the way DBT works makes it much trickier because no function calls can be made directly. Instead, QEMU provides the so-called helper and 'op' helper functions to cope with this issue.

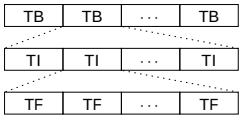


Fig. 3. TBC before insertion of IXC (cf. Fig. 4). Note that the lower layer is an enlargement of its immediate upper layer. In other words, the top layer shows that TBC is composed of an unknown number of TBs chained together. The second layer shows that each TB at the top layer is composed of an unknown number of TIs. The bottom layer shows that each TI in the second layer is composed of an unknown number of TFs. Or in short, TBC is composed of an unknown number of TFs.

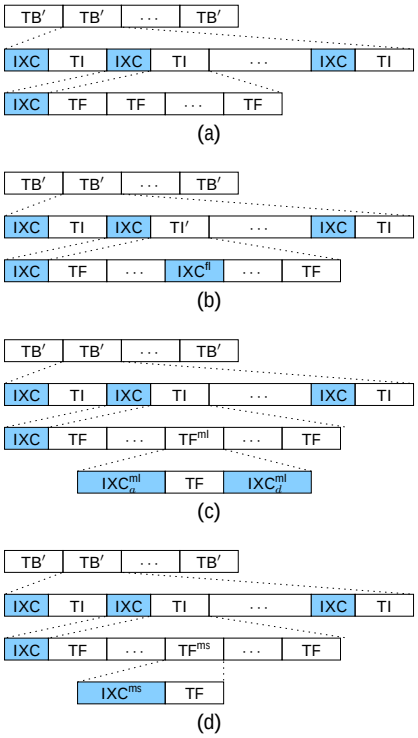


Fig. 4. TBC after insertion of IXC (cf. Fig. 3). Note that the lower layer is an enlargement of its immediate upper layer. (a) Non-load and non-store instructions, (b) Non-load and non-store instructions with condition code, (c) Memory load instructions, and (d) Memory store instructions.

To use the helper functions to extract the information SystemC needs for co-simulation, several things have to be done. To simplify our 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 the i-th parameter.

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

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

   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.
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. 4.
3) The third thing is to save all the parameters—such as the program counter of the next target instruction to be executed before it is translated to the host code—passed to the 'op' helper function away so that they can be treated as constant later on by the corresponding helper function. Similarly, each target instruction can be saved away and treated as a constant at run-time using exactly the same mechanism.

TABLE I
NOTATIONS USED IN FIGS. 3 AND 4

TB	Translation block.
TI	Target instruction.
TF	Tiny function.
IXC	Information extraction code.
TB′	TB with IXC.
TI′	TI with IXC.
IXC^{fl}	CPSR load IXC.
IXC^{ml}_a	Memory load address IXC.
IXC^{ml}_d	Memory load data IXC.
TF^{ml}	TF for memory load, i.e., with IXC^{ml}_a and IXC^{ml}_d.
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

Once we have all the helper functions for retrieving 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, as shown in Fig. 4. Note that the notations used in Figs. 3 and 4 are summarized in Table I. In what follows, we will discuss how the key information CA-ISS needs to provide are retrieved.

1) Simulating instruction fetch stage: Because the address of the target instruction and the target instruction itself will be discarded right after they are translated to the host code, their values have to be saved away so that they can be found later on at run-time. This is where the 'op' helper function kicks in. In other words, we can insert the 'op' helper function right after the place where the target instruction is being fetched so that the corresponding helper function will get called to send the address of the target instruction and the target instruction to SystemC, as shown in Fig. 4(a). The information necessary for simulating the pipeline of an ARM processor at the cycle-accurate level includes the program counter (pc) and the instructions located at pc, $\text{pc}+4$, and $\text{pc}+8$. Note, however, that since the offsets 4 and 8 are fixed, only pc needs to be sent to SystemC.
2) Simulating conditionally executed instruction fetch stage: The instructions that are conditionally executed can be easily detected because the code generator requires that the target processor instruction be decoded

first. Although the ISA of the target processor is quite different from that of the host processor, the way the condition codes are handled is basically the same, i.e., by checking the condition code flags inside the status register of the processor. As such, all we need to do is to insert a helper function to extract the status in the Current Process Status Register (CPSR) of the ARM processor either right before or right after the tiny function that checks the condition code, as shown in Fig. 4(b).

3) Simulating memory access stage: Since the memory access interface of QEMU version 0.10.x has been simplified, all we have to do 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. 4(c) and (d).

Since the sole purpose of the helper functions is to extract the address and data from the target instructions executed by QEMU instead of modifying their values, the behavior of executing the target instructions is guaranteed to be the same as though the helper functions do not exist, 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.

Upon receiving the information from QEMU, the cycle count of each instruction can be calculated, and the data dependency between instructions can be analyzed and synchronized with the system clock provided by SystemC. Because we choose the Versatile/PB926EJ-S as the experimental virtual platform, all the cycle counts comply with those specified in the ARM Architecture Manual [11] and ARM9EJ-S Technical Reference Manual [12].

The pipeline of the virtual processor can be simulated by translating the information provided by QEMU into the cycle-accurate waveform, as shown in Fig. 5, where pc indicates the program counter and $[\text{pc} + n]$ the instruction at $\text{pc} + n$ with n being a multiple of 4. Note that as far as this paper is concerned, the format of the information provided by QEMU is fixed and is referred to as *packet*, which is composed of pc, $[\text{pc}]$, $[\text{pc} + 4]$, and $[\text{pc} + 8]$.

The basic idea of providing the cycle-accurate information is to properly map information in packets to stages of the pipeline of a processor. Such a mapping depends, to a large extent, on the instruction and the processor in question. As such, the following discussions assume the processor under consideration is ARM. For the first instruction in booting up an OS and the instruction right after a branch instruction, we need all the information in a packet to properly map such an instruction to stages of the pipeline of a processor, which

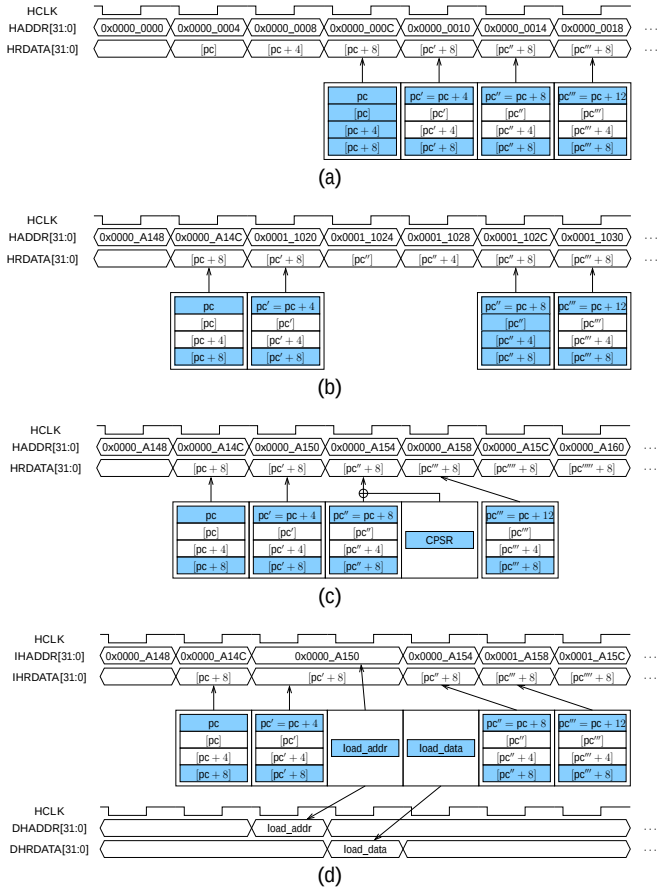


Fig. 5. Cycle-accurate waveform. See text for details.

is composed of three stages: fetch, decode, and execution, as shown in Fig. 5(a) and (b). For other instructions, we only need pc and $[\text{pc} + 8]$ in a packet to simulate the sequential addressing pipeline behavior. In Fig. 5(c), when decoding a conditionally executed instruction, two packets are required: the packet containing pc, $[\text{pc}]$, $[\text{pc}+4]$, and $[\text{pc}+8]$ followed by the packet containing CPSR. As for a single load instruction, three packets are required for decoding such an instruction. The first of which contains pc and instructions, the second of which contains the load address, and the third of which contains the load data, as shown in Fig. 5(d). Note that in this case, the address $\text{pc}+12$ on IHADDR[31:0] of the instruction bus is calculated from the packet containing the load address. This is required by SystemC to produce the correct waveform at the cycle-accurate level. Decoding a store instruction is similar to decoding a load instruction except that it requires two packets, the second of which contains both the address and data to be stored. Moreover, it requires two cycles to deal with the address and data on the data bus. As for the multiple load and store instructions, the number of packets required depends on the number of data, n, to be loaded and stored. As a rule of thumb, the number of packets for multiple load and store are, respectively, $1 + 2n$ and $1 + n$.

TABLE II
CYCLE-ACCURATE STATISTICS OF BOOTING UP THE LINUX KERNEL ON THE PROPOSED FRAMEWORK FOR 30 TIMES

Statistics	Co-simulation time	N_{TI}	N_{LD}	N_{ST}	N_{ILC}	N_{TC}	U_{IB}	U_{DB}	P_{DDI}
$\min$	16m06.448s	673,544,213.00	238,154,857.00	82,912,775.00	69,243,837.00	1,012,039,082.00	75.46%	31.78%	6.84%
		(67.72%)	(23.94%)	(8.34%)	(6.84%)	(100.00%)			
$\max$	17m50.171s	737,077,436.00	256,901,028.00	94,231,601.00	74,277,796.00	1,109,771,755.00	75.41%	31.77%	6.69%
		(67.73%)	(23.61%)	(8.66%)	(6.69%)	(100.00%)			
μ	16m36.490s	684,540,209.53	241,639,042.67	85,059,155.73	70,143,830.53	1,029,479,481.10	75.42%	31.80%	6.81%
		(67.69%)	(23.90%)	(8.41%)	(6.81%)	(100.00%)			
σ	00m25.536s	15,135,606.94	4,371,884.87	2,663,041.41	1,181,696.26	23,149,679.09	0.04%	0.03%	0.04%

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.
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_{ILC}	The number of interlock cycles of the virtual *ARM* processor.
N_{TC}	The total number of cycles of the target instructions executed.
U_{IB}	The instruction bus utilization of the virtual processor.
U_{DB}	The data bus utilization of the virtual processor.
P_{DDI}	The proportion of the data dependency between instructions executed.

B. I/O Interface

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 are modeled and constructed based on this I/O interface of QEMU. The I/O interface can be divided into two categories: PCI and memory-mapped I/O. They are suitable for different target processors. Our implementation of the I/O interface is similar in principle to that of QEMU-SystemC [5] except that the interrupt mechanism we provide is much more complete.

IV. EXPERIMENTAL RESULTS

In this section, we evaluate the performance of the proposed framework based on two different measures: (1) the time it takes to boot up a full-fledged Linux kernel, and (2) statistics that can be collected while the system is being boot up. We use the virtual Versatile/PB926EJ-S platform as the experimental virtual platform with a PrimeCell PL190 VIC modeled in SystemC. For all the experimental results given in this section, a 2.00GHz Intel Core 2 Duo T7300 processor machine with 1GB of memory is used as the host, and the target OS is built using the BuildRoot package [13], which is capable of automatically generating almost everything we need, including the cross-compilation tool chain, the target kernel image, and the initial RAM disk. The Linux distribution is Fedora 11, and the kernel is Linux version 2.6.30.8-64. QEMU version 0.11.0-rc1 and SystemC version 2.2.0 (including the reference simulator provided by OSCI) are all compiled by gcc version 4.4.1.

A. Time to Boot up Linux

The time it takes to boot up a full-fledged Linux kernel is shown in the column labeled "Co-simulation time" of Table II. The result shows that it takes less than 17 minutes to boot up a full-fledged Linux kernel on average. Even in the worst case, it takes no more than 18 minutes.

B. Cycle-Accurate Statistics

The cycle-accurate statistics are also shown in Table II, and the notations are defined in Table III. In Table II, the columns labeled "N_{TI}," "N_{LD}," and "N_{ST}" are as defined in Table III. They are given to show that the proposed CA-ISS can also provide the instruction-accurate information. Note that all the numbers given are, as the names of the rows suggest, the $\min$, $\max$, average, and standard deviation of booting up the ARM Linux and shutting it down immediately on the proposed framework for 30 times.

The percentages given in parentheses in the columns labeled "N_{TI}," "N_{LD}," and "N_{ST}" are computed as

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

where the subscript α is either TI, LD, ST. For instance, the percentage given in the column labeled "N_{TI}" of the row labeled "μ" of Table II is computed as

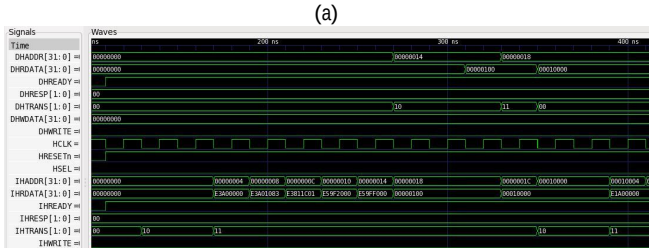
$$\frac{684{,}540{,}209.53}{1{,}011{,}238{,}407.93} \times 100 = 67.69\%.$$

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. As Table II shows, N_{TI}, N_{LD}, and N_{ST} count for about 68%, 24%, and 8%, respectively.

The cycle-accurate information are given in the columns labeled "N_{ILC}," "N_{TC}," "U_{IB}," "U_{DB}," and "P_{DDI}" and are as defined in Table III. More precisely, the column labeled "N_{TC}" in Table II is the total of internal cycles, non-sequential cycles, and sequential cycles of the processor's instruction bus as defined in ARM9EJ-S Technical Reference Manual[12]. Generally speaking, this value will be the same for both the instruction bus and the data bus. The column labeled "N_{ILC}" is the interlock cycles due to data dependency between instructions. Moreover, the internal cycles of the instruction bus or data bus indicate the idle state of the target processor. In summary, the percentage of the utilization of the instruction and data buses of the processor is calculated as

$$\frac{N_{\mathsf{TC}} - N_{\mathsf{B}}}{N_{\mathsf{TC}}} \times 100\%$$

(a)



(b)

Fig. 6. A snapshot of the waveform of AMBA on-chip bus. (a) QEMU-SystemC; (b) The proposed CA-ISS.

where N_{B} indicates either the internal cycles of the instruction bus or the data bus.

P_{DDI}—which indicates the proportion the performance of the system can be improved by getting rid of the data dependency between instructions—is computed as

$$\frac{N_{\mathrm{ILC}}}{N_{\mathrm{TC}}} \times 100\%.$$

C. Waveform of AMBA On-Chip Bus

In addition to the statistics given above, a snapshot of the waveform of the AMBA on-chip bus is given in Fig. 6. As Fig. 6(a) shows, QEMU-SystemC can only explore waveform of hardware models written in SystemC whereas the CA-ISS we proposed herein can not only trace waveform of hardware models written in SystemC but also all the instructions executed and all the memory accessed as shown in Fig. 6(b) using exactly the same AMBA bus protocol. Furthermore, the behavior of the pipeline of the processor can also be correctly modeled. Note that the waveforms given in Fig. 6 are shown by another open source tool called GTKWave [14].

More precisely, as Fig. 6(a) shows, the address jumps directly from 0x0000_0000 to the I/O read/write address of the virtual hardware device modeled in SystemC, and there is no information available for the instructions executed. As Fig. 6(b) shows, the address starts from 0x0000_0000 as indicated by the IHADDR signal and keeps increased by 4 until the load instruction reaches the execution stage at pc $+ 8$ (i.e., at 0x0000_0014 $=$ 0x0000_000C $+$ 0x8). Then, it will begin to load the data from the address 0x0000_0014. Moreover, IHRDATA reveals the sequence of machine code executed by the ARM processor when running a Linux kernel. Because the load instruction will alter the value of pc, the address will be changed to 0x0001_0000. Then, all the instructions pre-fetched into the pipeline from 0x0000_0014 up to 0x0000_0018 will be flushed, and the execution will continue. Note that the prefixes I/D indicate, respectively, the Instruction and Data bus signals of the ARM processor. Moreover, Harvard memory architecture separates the instruction bus from the data bus; thus, the instruction can be fetched from 0x0000_0018 of IHADDR[31:0], and the data can be accessed at 0x0000_0014 of DHADDR[31:0] simultaneously.

V. Conclusion

This paper presents a fast cycle-accurate ISS based on QEMU and SystemC for the SoC development. The CA-ISS proposed herein can not only produce the cycle-accurate waveform, but it also provides the capability of design space exploration. Moreover, our experience shows that the proposed framework can even benefit from whatever enhancements are made to QEMU and SystemC. Our experimental result shows that the design space exploration can be easily achieved by making QEMU play the role of an ISS, which will send all the information required by design space exploration to SystemC such as the instructions executed, the instructions fetched by the pipeline, the condition code flags in the processor register, the addresses of the memory accessed, the I/O operations performed, and so on. Finally, the "fast" co-simulation time indicates that the framework we proposed herein makes it possible to co-simulate a full-fledged operating system in the early stage of the SoC development.

Acknowledgment

The authors would like to thank the editor and anonymous reviewers for their valuable comments and suggestions on the paper. This work was supported in part by National Science Council, Taiwan, ROC, under Contract No. 98-2221-E-110-049.

References

[1] B. Bailey, G. Martin, and A. Piziali, *ESL Design and Verification*. Morgan Kaufmann Publishers, 2007.
[2] J. Chealier, O. Benny, M. Rondonneau, G. Bois, E. M. Aboulhamid, and F.-R. Boyer, "SPACE: A hardware/software SystemC modeling platform including an RTOS," in *Proceedings of Forum on Specification and Design Languages*, 2003.
[3] L. Benini, D. Bertozzi, A. Bogliolo, F. Menichelli, and M. Olivieri, "MPARM: Exploring the multi-processor SoC design space with SystemC," *Journal of VLSI Signal Processing*, vol. 41, no. 2, pp. 169–182, 2005.
[4] T. Givargis and F. Vahid, "Platune: A tuning framework for system-on-a-chip platforms," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 21, no. 11, pp. 1317–1327, November 2002.
[5] 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*, June 2007, pp. 2338–2341.
[6] Open SystemC Initiative (OSCI). [Online]. Available: http://www.systemc.org/
[7] *IEEE Standard System C Language Reference Manual*, 1666-2005.pdf, Design Automation Standards Committee, 2005. [Online]. Available: http://standards.ieee.org/getieee/1666/download/
[8] J. Bhasker, *A SystemC Primer*. Star Galaxy Publishing, 2004.
[9] F. Bellard. QEMU. [Online]. Available: http://bellard.org/qemu/index.html
[10] ——, "QEMU, a fast and portable dynamic translator," in *Proceedings of USENIX Annual Technical Conference*, June 2005, pp. 41–46.
[11] ARM, *ARM Architecture Reference Manual*, 2005. [Online]. Available: http://infocenter.arm.com/help/index.jsp
[12] ——, *ARM9EJ-S Technical Reference Manual*, 2002. [Online]. Available: http://infocenter.arm.com/help/index.jsp
[13] BuildRoot. [Online]. Available: http://buildroot.uclibc.org/
[14] GTKWave. [Online]. Available: http://gtkwave.sourceforge.net/