System-Level HW/SW Co-Simulation Framework for Multiprocessor and Multithread SoC

Moo-Kyoung Chung, Sangjun Yang, Sang-Heon Lee and Chong-Min Kyung
Department of EECS, KAIST
373-1, Yusong-gu, Daejeon, Korea
mystery@vlab.kaist.ac.kr

ABSTRACT

C/C++-based languages such as SystemC or SpecC can be used for both hardware and software description by raising the level of abstraction for hardware. This paper proposes techniques for fast and accurate high-level co-simulation for multithread and multiprocessor SoC design using SystemC for hardware and legacy C with RTOS (Real-Time Operating System) API for software. Automatically modified legacy C synchronizes with SystemC clock events, and communicates with other modules through I/O(Input/Output) variables and transaction level bus models. Generic RTOS scheduler and POSIX APIs are also provided for the real-time application. About three times faster co-simulation speed than the ISS-based co-simulation along with various profiling data with 95% accuracy were achieved.

1. INTRODUCTION

Co-simulation that offers united simulation platform for the hardware and software is essential for a complicated SoC(System on Chip) design in exploring system architecture and debugging. In addition, system design and verification at higher levels of abstraction are important to meet the time-to-market pressure.

SystemC is a C++ class library for hardware description and simulation. Designers, therefore, can use C/C++ language for modeling hardware as well as software and can easily describe the hardware at high-level of abstraction by means of SystemC and C/C++ behavioral modeling. There are, however, difficulties in simulating the software that is executed on the target processor, because SystemC does not have any processor models. There are two methods to co-simulate the target software with the SystemC module. One is to use the target processor model and the other is to use native code execution. For the former method, designers should have either SystemC model for the target processor or ISS(Instruction Set Simulator) with communication channel between ISS and SystemC model. While the latter method is easy to use in the high-level model and has much faster simulation speed, designers should solve the following problems to support the legacy C for the software model.

- how to let software execution synchronize with hardware clock event
- how to let software model communicate with hardware model
- how to support RTOS with SystemC

This paper introduces co-simulation techniques that make the software model written in legacy C synchronize and communicate with the hardware model written in SystemC, and explains how to automate the modeling. This paper also explains how to model real-time scheduler using SystemC and how to support multithread software with POSIX real-time API.

The rest of this paper is organized as follows. Section 2 discusses related works. Section 3 explains about techniques for synchronization, communication and RTOS. Section 4 describes automatic augmentation of the software model and other features. The experimental results are presented in section 5.

2. PREVIOUS WORKS

There are many researches on co-simulation using SystemC for the hardware simulation and ISS for the target processor, [1], [2], [3] and [5]. Fig. 1 illustrates these approaches together. There are two kinds of modeling methods. First, SystemC and each ISS have their own processes in the host OS, and they communicate with each other using IPC(Inter-Process Communication) shown in Fig. 1(a), (b) and (c). Second, the co-simulation is performed on single SystemC process in which the ISS is embedded. Each model communicates through SystemC ports and signals shown in Fig. 1(d). Both methods are difficult to implement and apply for the abstracted software model, and they have simulation time overhead of ISS. The former method has larger IPC overhead.

In order to solve these problems, the native code execution of legacy C for the software model is presented in [4] and [6]. [4] introduces `delay()' function for the synchronization between legacy C and SystemC, and [6] makes `legacy-code global variable’ class template for the communication. They speed up the co-simulation through the native code execution. Users, however, should insert synchronization functions into their software manually. It is inaccurate, since users cannot know the clock-cycle consumption of the software before compiling and running the software for the target processor. It also causes long synchronization period, because the synchronization functions are inserted at function boundaries of the C code. Besides, it is difficult to support various C coding styles and hardware communication protocols only using the ‘legacy-code global variable’.

While the proposed technique is based on these methods, it overcomes the above mentioned restrictions.

Nowadays, embedded systems, for the most part, require multithread programming and real-time capability. There has been increasing interest in RTOS simulation on the SystemC environment such as [7], [8], [9], [10] and [11]. Fig. 2 illustrates how RTOSs are supported in these works. (a) has ISS overhead. (b) shows that the host OS simulates the threads and thread scheduler of the target processor, but its behavior cannot be the same as the target RTOS.
Both (a) and (b) have IPC overhead. For (c) and (d), users should develop the software using SystemC API. (c) supports RTOS scheduling by SystemC kernel modification, while (d) focuses on developing the software using SystemC API. (c) supports RTOS execution, while (d) supports RTOS execution with POSIX API.

3. CO-SIMULATION

The proposed co-simulation framework is presented in Fig. 3. A processor module consists of augmented user code, IO variables, and RTOS block. The augmentation denotes annotating the synchronization function into user code and expanding the code into software model of the target processor. Each block synchronizes with other processors or hardware modules using the synchronization function. IO variable takes the role of communication through TLM (Transaction Level Modeling), and a real-time scheduler schedules the threads on the processor.

A. Synchronization

A SC_CTHREAD of software model calls user code, and every statement of the user code is annotated with a synchronization function, thread_sync(). The thread_sync(cycle_consumption) function waits for cycle_consumption that is the target processor clock-cycles consumed by the statement of the user code. The cycle_consumption is calculated by using the corresponding instructions in the target executable file, which is the compilation outcome of the target cross-compiler. This process is automated by Augmentor, which will be explained in section 4-A.

Fig. 4(a) is a software class derived from a target processor class and a generic RTOS class. A SC_CTHREAD in the software class calls user_main function of the augmented code, shown in Fig. 4(b), where the user_main originates from main function. The thread_sync function is inserted ahead of every statement of the user code that consumes clock-cycles of the target processor. The function waits for the corresponding clock-cycles using a wait() function of SystemC. If 50MHz clock is used for the target processor and 100MHz for hardware clock, thread_sync(2) calls wait(4). The clock-cycles of the IO access instructions are taken into account by the IO variable class through TLM.

The cycle_consumption of C statement comes from the cross-compiled executable code. However, some instructions generate computational loop whose cycle count is not determined at static compile time but depends on the instruction operands whose values are dynamically changing. We made a member function of the target processor class that calculates the cycle counts of those instructions at run-time using the information about the target instructions and operands. Underlined statement of the Fig. 4(b) calls the member function.

B. Communication

IO variable class templates are provided to ease the communication among modules. Users do not need to rewrite their code, because the IO variable classes are derived from TLM class and have member functions to access the TLM using the operator overload of C++ for all the possible operators. Augmentor exchanges the declarations of the variables expected to communicate with other modules for the declarations of corresponding IO variables.

Fig. 5 shows a simple example of the calling TLM from the user code. If an integer variable ‘a’ is to access an IO component, which is located at 0xC0000000 address space, Augmentor replaces the declaration of the variable with the third line of Fig. 5(a). The access from the user code initiates the TLM through the member functions of the IO variable class shown in Fig. 5(b).

In order to simulate HW/SW communication at high abstraction level, we employed transaction level bus model. TLM makes simulation fast with high-level abstraction without detailed pin-based implementation while keeping the exact clock-cyclic simulation of operations. We obtained fast and accurate simulation of communication by using the target bus model derived from generic TLM.
We support the following kinds of descriptions for the IO variables: fundamental data type, array, pointer, structure, and file. They cover all the coding styles for access to IO components. Table I shows the IO variable replacement in detail.

There are several types of transfers in a bus protocol, such as single transfer, burst transfer, etc. In legacy C, since IO access code has no information about the transfer type, the simulator cannot make a decision on which to choose among the transfer types. We solve the problem by getting the information from the target instructions. It is possible, since the simulator knows the target instructions corresponding to the IO access statement. We exchange the IO access statement for the equivalent statement that calls the TLM function appropriate to the transfer type. Users can also edit the statement using TLM function in order to let desired type of transfers occur.

C. RTOS

The software class is derived from both generic RTOS API class and real-time scheduler class. The generic RTOS API is a subset of the POSIX 1003.1c and POSIX 1003.1d real-time extension, and the real-time scheduler handles the user threads that will be executed on the POSIX 1003.1c and POSIX 1003.1d real-time extension, and the real-time scheduler class. The generic RTOS API is a subset of operator overloading.

The scheduler_thread updates the states of user threads every timer tick and carries out the sequential scheduling while assuring that two or more threads do not have running state at the same timer tick period. The time consumption in the RTOS scheduler and context switching is also considered by the scheduler_thread shown in Fig. 6(d). The time consumption may depend on the number of threads or scheduling policy. The RTOS class lets users modify the time consumption calculation function using the virtual member functions of C++. The scheduler_thread checks events from TLM in every clock-cycle in order to support hardware interrupt. If an interrupt occurs, scheduler invokes corresponding interrupt service routine.

There is a problem in applying SC_CTHREAD to simulate user threads, because the SystemC thread could not be allocated dynamically. It should be possible to create or close user threads at run-time. We solve this problem with pre-defined SC_CTHREADs. Augmentor makes the SC_CTHREADs in the augmented user code as many as the maximum number of user threads at static time, where the number is informed by users. Before a SC_CTHREAD is allocated to the user function at run-time, it does nothing while executing wait() function loop.

4. IMPLEMENTATION

A. Augmentor

The functions of Augmentor are as follows:
(a) To annotate synchronization code into the user C code using the cycle count information from the cross-compiled ELF file
(b) To exchange the declaration of the variable that accesses hardware for the declaration of the IO variable class template
(c) To generate processor class that encapsulates the generic RTOS class, IO variable class and annotated user code

Fig. 7 shows the flowchart of Augmentor. Augmentor extracts the instructions that correspond to a statement of the user code from the text section of ELF based on the debug information, which is in debug sections with DWARF, and then calculates the cycle count of the target instructions. This technique is worthy of use independent of the compiler optimization, because the debug option (‘-g’ for gcc) makes the target cross-compiler generate the target executable with debug information regardless of the optimization option (‘-O’ for gcc.)

It is possible to apply the augmentation of software model to various processors and bus architectures. Augmentor could support another processor by making some member function of the processor class, instruction decoder and instruction clock-cycle table if the target cross-compiler would be available. Since OOP(Object Oriented Programming) is employed for modeling the system, various bus architectures are supported by changing the properties or making the virtual functions of the target TLM.
B. Debugging and Analysis

Since the proposed co-simulator runs on the single SystemC process, it is easy to debug and analyze software and hardware simultaneously. Users can use C++ source level debugger to debug not only the SystemC hardware module but also the software model with original user code without being troubled by the annotated lines, since Augmentor annotates the synchronization functions with the line numbers of the original user code. The real-time scheduler writes the state transition history of each thread and the thread switching overhead on the trace file using SystemC trace generation APIs. The result waveform, therefore, shows the scheduler behavior and thread transition timing with hardware signals transition diagram as shown in Fig. 8.

We explored the architecture of a JPEG decoder using the proposed framework. The target processor is ARM9 and the host processor is Intel Xeon 2GHz processor with Linux OS. Fig. 9 shows the JPEG decoding sequence and the block diagram of the target architecture. Three architectures are experimented: (Case-A) both VLD and IDCT block are in hardware; (Case-B) VLD in software, IDCT in hardware; (Case-C) VLD in hardware, IDCT in software. We compared the proposed co-simulator with the ISS and IPC based co-simulator as shown in Fig. 1(b).

<table>
<thead>
<tr>
<th>Technique</th>
<th>Architecture</th>
<th>CPU Cycle Count</th>
<th>Simulation Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISS, IPC-based</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Case-A</td>
<td>2,303M</td>
<td>100%</td>
<td>723</td>
</tr>
<tr>
<td>Case-B</td>
<td>2,934M</td>
<td>100%</td>
<td>798</td>
</tr>
<tr>
<td>Case-C</td>
<td>3,557M</td>
<td>100%</td>
<td>853</td>
</tr>
<tr>
<td>Proposed</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Case-A</td>
<td>2,197M</td>
<td>95.4%</td>
<td>281</td>
</tr>
<tr>
<td>Case-B</td>
<td>2,781M</td>
<td>94.8%</td>
<td>291</td>
</tr>
<tr>
<td>Case-C</td>
<td>3,359M</td>
<td>94.4%</td>
<td>310</td>
</tr>
</tbody>
</table>

TABLE II. Result for the JPEG decoder

We were able to speed up co-simulation about three times and achieve 95% accuracy of the performance estimation for the target application. Besides, waveform file can be obtained that shows both hardware signal transitions and the software behavior at a drawing. TABLE II and Fig. 8 show the co-simulation results and the waveform.

6. SUMMARY AND CONCLUSIONS

In the proposed simulator, the native code execution of legacy C, SystemC modeling and TLM are used for the software, hardware and communication model, respectively. These models work together on the SystemC single process of the host OS. We proposed fast and accurate co-simulation techniques including detailed implementation skills, as focusing on the software modeling techniques for the HW/SW synchronization, communication and RTOS simulation with their automatic implementation. Designers can easily apply the co-simulation framework to SoC design without troublesome re-writing or refinement to lower abstraction level. The simulation speedup in the experimental result shows the effectiveness of the techniques.

REFERENCES