# System-Level Development and Verification Framework for High-Performance System Accelerator

Chen-Chieh Wang, Ro-Pun Wong, Jing-Wun Lin, and Chung-Ho Chen
Institute of Computer and Communication Engineering
National Cheng Kung University, Tainan, Taiwan

## ABSTRACT

In this paper, we propose a framework to develop high-performance system accelerator at system-level. This framework is designed by integrating a virtual machine, an electronic system level platform, and an enhanced QEMU-SystemC. The enhancement includes a local master interface for fast memory transfer, and an interrupt handling hardware for software/hardware communication support that enables full system simulation. We have also developed a network virtual interface for our system to co-work with the real world network environment. Finally, the MD5 algorithm offload and the network offload engine are used as examples to demonstrate the proposed framework system for full system simulation.

## 1. INTRODUCTION

Application specific computer systems are seeking higher performance to satisfy a variety of applications. In the past, the emphasis is primary on the promotion of single core performance while now the attention may focus on a multi-core system. A system that offloads tasks to a high-performance subsystem from the host can be regarded as another kind of multi-core system. This architecture not only minimizes the burden of the host processor, but also enhances the performance of the whole system by embedding specialized hardware accelerator into the system. In this paper, we call such a subsystem a "system accelerator." Examples of such systems can be found in cryptographic accelerator, 3D graphics, or high-speed network applications where it is often inefficient to use only generalized CPUs. Instead, a specialized system accelerator implemented in the Host Bus Adapter (HBA) is used to enhance the system performance.

A computing system which composes of a processor system and a system accelerator in HBA, as shown in Figure 1, not only minimizes the burden of the host CPU by offloading heavy tasks to the HBA, but also may provide scalability for either higher throughput or more computation power.

The development of System-on-a-Chip (SoC) technology makes an HBA become a specialized and often complicated computing system that contains internal processors, specialized hardware accelerators, memory systems, and I/O interfaces. In traditional design and verification flow, the implementation of HBA's internal system must complete before a software/hardware operational environment that composes of a host processor, operating system, and an FPGA verification platform can be built up. Thereafter the device drivers and related applications are developed in order to complete prototyping verification for the entire system.

Since the internal system in current HBA subsystems may be as complicated as that of an SoC design, we can therefore apply Electronic System Level (ESL) design and verification methodology [1] to shorten the development time. ESL design aims to model the behavior of the entire system using a high-level language such as C/C++ or SystemC [2], and introduces new concepts such as Transaction Level Modeling [3], and Event Driven Modeling.

Current ESL development environment, e.g. CoWare Platform Architect [4] or SoC Designer [5], puts emphasis on SoC design, consisting of microprocessor, memories, and interconnection

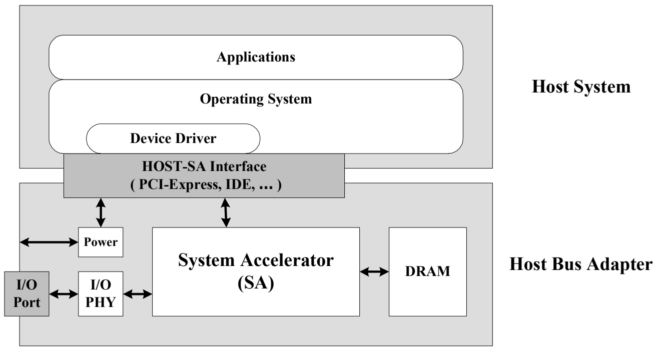

Figure 1. A system accelerator in Host Bus Adapter

modules among processing units. To develop a system accelerator in HBA, the interaction between the host and the accelerator subsystem, along with the partition of application tasks and the efficiency of the interactions, must be taken into consideration.

In order to model the host system, we can make use of the microprocessor, which can be an instruction-set simulator or a cycle-accurate model, provided by the ESL platform to play the role of the host CPU. However, such architecture can only be used to simulate the interaction between the host CPU and the system accelerator through simple control programs. If a real world operating system is required, the speed of simulation may be unacceptable.

In the development of a system accelerator, a full-system environment is preferred, such as Simics [6], M5 [7], or QEMU [8]. A full system simulation platform (including processor cores, peripheral devices, memories, interconnection buses, and network connections) is able to boot and run an unmodified commercial operating system. It can also run realistic workload under a reasonable simulation time. Nevertheless, current support on developing virtual hardware for full system simulation platforms is not as rich as that found in commercial ESL integrated development environment.

In this paper, we integrate a virtual machine with an ESL integrated development environment, to provide a fine-grained system-level development and verification framework for high-performance system accelerator.

This paper is organized as follows. Section 2 and Section 3 describe the system architecture and framework respectively. Section 4 discusses the implementation issues. Section 5 describes two examples to demonstrate our system. Finally, Section 6 concludes this paper.

## 2. SYSTEM ARCHITECTURE

Figure 2 shows the system architecture of the simulation framework that consists of a virtual machine platform and the intended subsystem accelerator. The host system is modeled with the virtual machine whereas the system accelerator can be developed using the related tools provided by the ESL design kits such as [4], [5].

## 2.1 Host System

A good simulation is a trade-off between its accuracy and its performance, i.e. simulation time. By simulating only the necessary details, we can simulate effectively while still reaching a certain acceptable accuracy. Our main target is to build a system accelerator development environment, so what is required for the host, is an open-source operating system, and a tool-chain to develop the device drivers. Due to this need, we have used an un-timed virtual machine model to provide full system capability and at the same time to accelerate the simulation speed for the host system. To achieve this, we have chosen QEMU as our virtual machine. QEMU is an open source virtual machine, and is used in many projects, such as Google Android [9]. We modify the QEMU source code and build the required hardware as un-timed models. If a timed model is required, a project, call QEMU-SystemC [10][11] can provide us for such a design environment.

## 2.2 System Accelerator

Current commercial ESL integrated development environment can be used as a system accelerator development environment. For example, CoWare Platform Architect, SoC Designer, or some other ESL development platforms. These ESL development platforms support SystemC and C/C++ modeling tool, and some even support co-simulation between SystemC and Hardware Description Language (HDL).

Designers can use all kinds of microprocessor, on-chip-bus, peripherals, and other built-in libraries to build the SoC components in the system accelerator. The behavior model of I/O devices and DRAM can also be developed using SystemC or C/C++. In simulations, the power system is not necessary, but it is possible to include power estimation techniques into ESL models for evaluation. The details of the interactions between a host and a system accelerator will be discussed in the later section on implementation.

## 3. System Framework

To demonstrate full system simulation capability of the proposed framework, we use a network offload engine as an example to describe the system framework and design flow.

## 3.1 Functional Verification

According to the top-down concept in ESL design flow, high level model is adopted to implement the entire system in the preliminary stage. Then Golden Testbench is set up for use in the following development and verification stages.

For simplicity, we refer the host system in QEMU emulation as "Virtual Host," and the host system in the real computer as "Physical Host." As shown in path 1 in Figure 3, we can build *Virtual Host 1* in *Physical Host 1* (run on *Computer 1*) by QEMU, and an HBA simulation environment in ESL platform is connected to *Virtual Host 1*. As we know that network connections are paired architecture, if our network offload engine comprising *Virtual Host 1* with its HBA, is capable of communicating with *Physical Host 2* (run on *Computer 2*) in another physical environment, we can be sure that in functionality the network offload engine performs equivalently to the system which has not offloaded the network protocol.

In this way, we can develop the network offload engine in the ESL platform using a high level model, and define the interface for interactions between the HBA and its host, which allows the device drivers and related testbench to be designed and tested at an early stage. As for how the network packets are transferred between the virtual I/O interface and the physical I/O interface, the details will be described later in the implementation section.

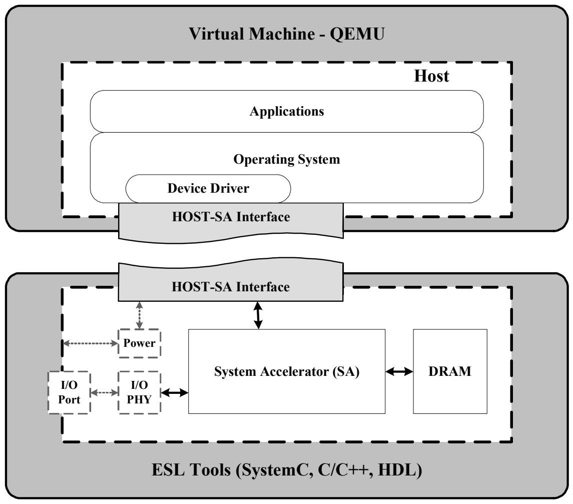

Figure 2. System architecture

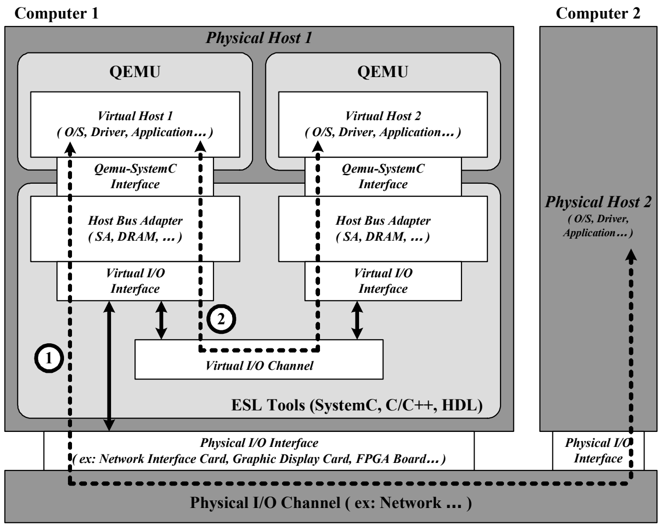
Computer 1
Computer 2
Physical Host 1
QEMU
QEMU
Virtual Host 1 ( O/S, Driver, Application… )
Virtual Host 2 ( O/S, Driver, Application… )
Qemu-SystemC Interface
Qemu-SystemC Interface
Host Bus Adapter ( SA, DRAM, … )
Host Bus Adapter ( SA, DRAM, … )
Virtual I/O Interface
Virtual I/O Interface
2
1
Virtual I/O Channel
ESL Tools (SystemC, C/C++, HDL)
Physical Host 2 ( O/S, Driver, Application… )
Physical I/O Interface ( ex: Network Interface Card, Graphic Display Card, FPGA Board… )
Physical I/O Interface
Physical I/O Channel ( ex: Network … )

Figure 3. System framework

## 3.2 Performance Evaluation

With the Golden Testbench, we can use a more accurate model to implement the system accelerator and continue with the top-down design flow on the ESL platform.

Specifically, most network protocols or applications use handshaking or bi-directional data transfer. However, the HBA in *Virtual Host 1* has a different clock domain when communicating with *Physical Host 2* (the clock domain in the ESL platform is much slower than that in real world environments). This leads to the inaccuracy performance estimation for the system accelerator in HBA. Therefore we need another framework, shown in path 2 in Figure 3, to overcome this clock-domain mismatch.

In this framework, we use QEMU to set up *Virtual Host 2* in *Physical Host 1*, and duplicate the verified HBA simulation environment (connected to *Virtual Host 1*) and associate it with *Virtual Host 2*. They communicate with each other through the virtual I/O channel. In this case, the two HBAs are in the same ESL platform, and therefore have identical clock domain. Consequently we can evaluate the performance of the system accelerator in HBA. There is an additional advantage when the virtual I/O channel can be parameterized for simulation. Taking our network offload engine as an example, we can directly control the bandwidth and latency in this virtual network.

The only limitation in our platform is that there is no timing information in the host, so it is hard to precisely model the latency taken by the host CPU to run a program. A Host-HBA bridge, however, has been implemented in the ESL platform, which can be used to model the bandwidth of the communication channel between HBA and the host. Hence we are still able to estimate the performance of system accelerator in HBA.

A possible solution can address to this limitation. We can profile the events between HBA and the host, such as an interrupt service routine, in functional verification stage. Then the timestamp is measured by executing these events in real computer or counting its dynamic instruction count. Finally, we can evaluate the performance of the whole system by embedding the timing information into Host-HBA bridge module to create appropriate latency.

## 4. IMPLEMENTATION ISSUES

Figure 4 shows the detailed structure for our implementation that uses CoWare Platform Architect for ESL and/or RTL development.

### *4.1 Operating System and Device Driver*

Linux is chosen as our operating system. Since Linux is an open source environment, it has much flexibility for us to make changes on the host operating system environment. For different operating systems and different system accelerators, specific device drivers are required.

Common device drivers use either I/O register access or memory access to communicate with the hardware. We have used the specific APIs to provide these accesses, which provide easy modification that can be ported to different system accelerators. We have also provided a standard interrupt service routine, which provides a response to the hardware suitable for our designated interrupt handler.

### *4.2 The Interface between Host and System Accelerator*

To make our system a general environment, that suits most needs, we have upgraded the interface in QEMU-SystemC, including PCI bus adapter, which applies to x86 environment, for studies in desktop systems, and AHB bridge, which applies to AMBA environment, for studies in embedded systems.

The QEMU-SystemC project has provided QEMU a data transfer interface to access data from the SystemC module. However, it has not yet provided a complete communication mechanism for software and hardware.

Typical communication mechanisms for software and hardware are polling, interrupt, or a combination of both, and require both software and hardware support. The Linux device driver model has provided software support, like polling APIs and interrupt service routine, and the system accelerator in the ESL environment requires hardware support. Since polling in implementation is just reading the memory, it requires no extra hardware support. For interrupt, an interrupt controller is required. Therefore, an interrupt controller is implemented along with the system accelerator, and an interrupt line is connected from the system accelerator to the internal interrupt handler in QEMU. In this way, the system accelerator is able to send interrupts to the host QEMU, and ask the interrupt service routine in device driver to respond to this event.

In certain applications, such as network offload engine, which needs to transfer a large amount of data to the host system, interrupt is a barrier to high speed transfer. Therefore, an interface that accesses host memory from the system accelerator is used to provide faster speed. Due to this reason, we have designed a local master interface in HOST-SA Bridge module, to satisfy such needs.

In a system accelerator, many kinds of interfaces can be adopted to connect the HOST-SA Bridge module and Interrupt Controller

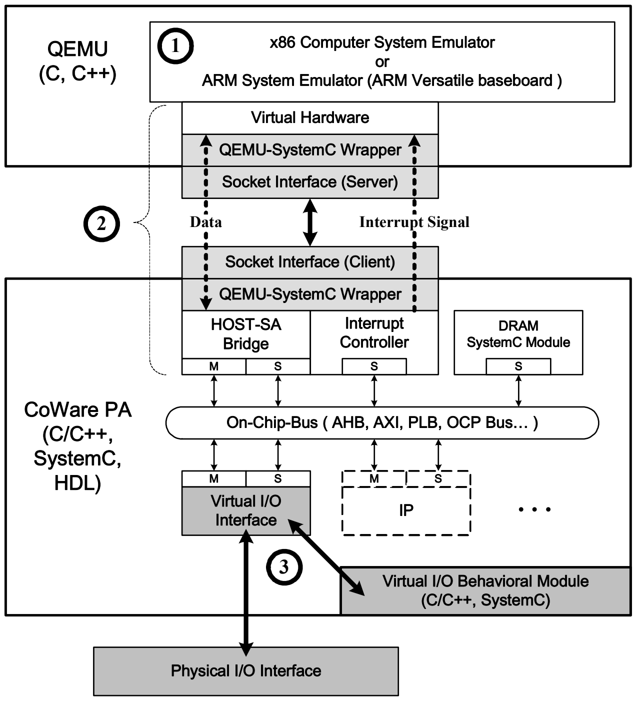

Figure 4. Detailed implementation

module with different on-chip-bus. In order to set up an AMBA-based system, we use the API of CoWare AHB TLM to attach these modules to the AHB bus. In addition, Open Core Protocol (OCP) [12] can also be used to carry out a more flexible interface that is compatible with other types of ESL platforms.

### *4.3 Virtual Input/Output Interface*

The virtual I/O interface function provides us a verification interface to connect to either a virtual behavior module, or a physical I/O module. For network issues, it allows designers to connect to the real world computer through network, or a virtual network testbench. For graphic issues, it allows designers to connect to a virtual or physical monitor.

To connect to the physical environment on the network, we have used RawIP [13], a library, to send a self made MAC frame or IP datagram to the real world network interface card and bypass the normal TCP/IP flow in the operating system. The network offload engine can also receive the MAC frames directly from the network interface card. This provides a channel for the HBA to communicate with a real computer using MAC frames and interact with the outside world.

The advantage of the virtual network is that we can simulate a 10Gbps or even 100Gbps network, and measure the system performance without acquiring a real network interface card.

### *4.4 Communication Mechanism*

The communication mechanism between QEMU and SystemC can be classified as either working in a distributed system or in a single computer. For a distributed system, network sockets can be used to communicate between processes in different computers. In this way, the system can be expanded easily, but requires higher overheads for each communication. For a single computer, kernel process communication mechanisms, such as shared memory, can be used. In this way, the system has lower communication overheads, but is less scalable. Both of the versions have been developed, and can be applied to different use of environments.

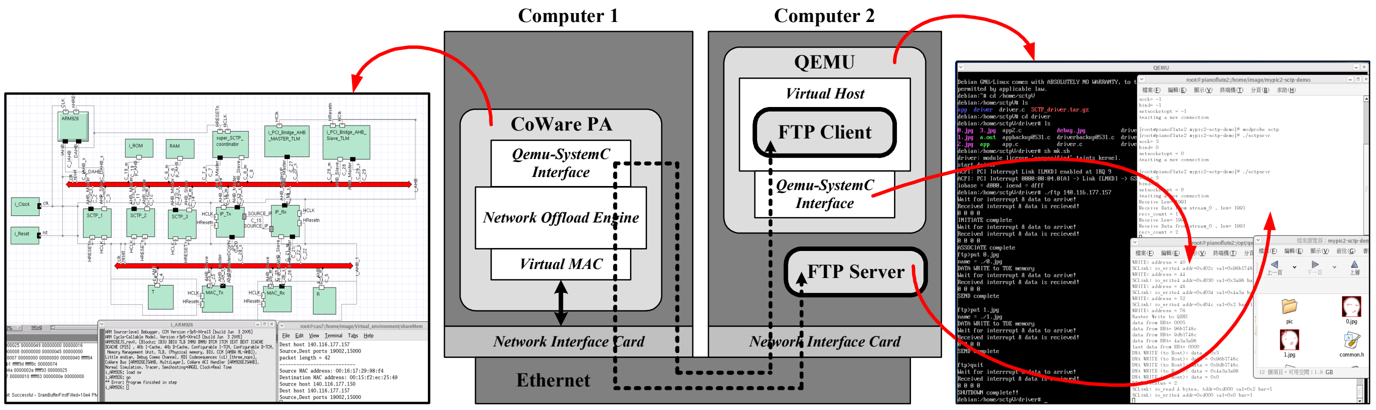

Figure 5. Communication between a virtual platform and a real world computer

## 5. Platform Demonstration

We have made two case studies with our platform. First, we implemented an MD5 offload engine to verify the accuracy of the interface between the host and system accelerator in the platform, including data transfer and interrupt mechanism (the second part in Figure 4). We offloaded the tasks of the instruction "md5sum" in Linux (a hashing function for authentication) to the MD5 hardware module in HBA. The detailed steps are as follows:

(1) When the driver of MD5 offload engine is invoked, the file to be processed with MD5 checksum was transferred from QEMU to the internal memory of HBA through QEMU-SystemC interface.

(2) The driver of the upper layer commands the MD5 hardware module in HBA to calculate the checksum of that file.

(3) The MD5 hardware module informs the upper-layer QEMU of the completion of the task through interrupt.

(4) At last QEMU retrieves the 128-bit result from the HBA and prints it out.

Secondly, we have developed a network offload engine onto our platform. We integrated the network offload engine with the device driver, and developed a network example that connects the real world network to our platform, running a simple FTP application to transfer data.

Figure 5 shows the implementation architecture of this example. For load balance, we have used two computers to run CoWare Platform Architect and QEMU, respectively. The FTP server is executed by the Computer 2, and the FTP client is executed by the virtual machine that is emulated by QEMU in Computer 2. The network offload engine that is executed by CoWare Platform Architect in Computer 1 has offloaded the network layer and transport layer of the network protocol stack from virtual machine. Therefore, the FTP client in virtual machine can communicate with real network environment by calling the APIs for network offload engine.

Figure 5 also shows the run-time status of this example. The FTP client in the virtual platform was uploading files to the server. The FTP server in the real world computer was receiving data from the client. Finally, the files had been received completely at the server.

## 6. Conclusions

In this paper, we have enhanced the work of QEMU-SystemC, including an enhancement between host and system accelerator communications, a fast transfer support, and a connection to physical world. Through this work, we have developed a framework that provides a full system simulation ESL environment for high-performance system accelerator designs. We also showed two case studies in order to demonstrate the practicability of this framework. Designers can rapidly develop the subsystem hardware or IPs in the ESL environment and related software including device drivers in the virtual machine for full-system hardware-software co-simulation. This paper focuses on HBA design; however, our framework can also be used for other specific system accelerators or IPs in embedded systems.

## Acknowledgment

This work was supported in part by the Industrial Technology Research Institute (ITRI) and the National Science Council, Taiwan, under Grant NSC 96-2221-E-006-192-MY3.

## References

[1] Brian Bailey, Grant Martin, and Andrew Piziali, "ESL Design and Verification: A Prescription for Electronic System Level Methodology," Morgan Kaufmann/Elsevier, 2007.
[2] Open SystemC Initiative (OSCI), http://www.systemc.org/.
[3] Dan Gajski and Lukai Cai, "Transaction Level Modeling: An Overview," HW/SW Co-Design Conference (CODES), 2003.
[4] CoWare Platform Architect, http://www.coware.com/products/platformarchitect.php.
[5] SoC Designer, http://carbondesignsystems.com/.
[6] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A Full System Simulation Platform," IEEE Computer, vol. 35, Iss. 2, pp. 50-58, Feb. 2002.
[7] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt, "The M5 Simulator: Modeling Networked Systems," IEEE Micro, vol. 26, Iss. 4, pp. 52-60, Jul.-Aug. 2006.
[8] QEMU, http://bellard.org/qemu.
[9] Android - An Open Handset Alliance Project, http://code.google.com/android/.
[10] QEMU-SystemC, GreenSocs, http://www.greensocs.com/en/projects/QEMUSystemC.
[11] Monton Marius, Portero Antoni, Moreno Marc, Martinez Borja, and Carrabina Jordi, "Mixed SW/SystemC SoC Emulation Framework," IEEE Int'l Symp. on Industrial Electronics, 2007.
[12] OCP International Partnership, http://www.ocpip.org/.
[13] A brief programming tutorial in C for raw sockets, http://mixter.void.ru/rawip.html.