Technical Report

Integration of Virtual Platform Models into a System-Level Design Framework

Pablo E. Salinas Bomfim and Andreas Gerstlauer

Computer Engineering Research Center

The University of Texas at Austin

UT-CERC-10-02

August 30, 2010

Computer Engineering Research Center
The University of Texas at Austin

1 University Station, C8800
Austin, Texas 78712-0323

Telephone: 512-471-8000
Fax: 512-471-8967

http://www.cerc.utexas.edu

THE UNIVERSITY OF TEXAS AT AUSTIN

UT ECE

ELECTRICAL & COMPUTER ENGINEERING

Integration of Virtual Platform Models into a System-Level Design Framework

Pablo E. Salinas Bomfim and Andreas Gerstlauer

Computer Engineering Research Center

The University of Texas at Austin

Abstract

The fields of System-On-Chip (SOC) and Embedded Systems Design have received a lot of attention in the last years. As part of an effort to increase productivity and reduce the time-to-market of new products, different approaches for Electronic System-Level Design frameworks have been proposed. These different methods promise a transparent co-design of hardware and software without having to focus on the final hardware/software split.

In our work, we focused on enhancing the component database, modeling and synthesis capabilities of the System-On-Chip Environment (SCE). We investigated two different virtual platform emulators (QEMU and OVP) for integration into SCE. Based on a comparative analysis, we opted on integrating the Open Virtual Platforms (OVP) models and tested the enhanced SCE simulation, design and synthesis capabilities with a JPEG encoder application, which uses both custom hardware and software as part of the system.

Our approach proves not only to provide fast functional verification support for designers (10+ times faster than cycle accurate models), but also to offer a good speed/accuracy relationship when compared against integration of cycle accurate or behavioral (host-compiled) models.

Table of Contents

List of Tables

List of Figures

CHAPTER 1

Introduction

The fields of System-On-Chip (SOC) and Embedded Systems Design have received a lot of attention in the last years. As part of an effort to reduce the time-to-market of new products and increment productivity, different approaches for System-Level Design have been proposed [1][2]. These different methods promise a transparent co-design of hardware and software without having to focus on the final hardware/software split.

One fundamental problem in System-Level Design (SLD) is the complexity and heterogeneity of the systems. This is especially true in the design of Embedded Systems. Therefore, in order to model complex systems, they are hierarchical decomposed in simpler models of computation and communication.

In an integrated approach, one set of tools should be used for all models, and the synthesis task should merely transform one level of system abstraction into a more detailed representation (e.g. from RTL into a netlist).

At the highest level of abstraction, hardware and software elements of a system should be represented in a form that successively decouples the specification from the implementation. This representation is a formal model of the computational and communication blocks that encompass the whole system being developed. Some models of representation have been syntactically formalized as System-Level Design Languages (SLDLs).

Besides the capability of having one (or several) models as an input to the refinement process, SLD tools need to include the ability of performing hardware exploration and fast functional verification through the mapping of computational elements to processor simulation models. Likewise, the SLD framework should have a database of communication models at a pin-accurate and/or transaction level for mapping and simulation of busses and communication mechanisms.

The above can be found as part of some commercial or freely available Electronic System-Level (ESL) tools. The System-On-Chip Environment (SCE) [9], developed at the UC-Irvine, is one example of this sort of tool. Domer et al [9] have proved that SCE can be used to achieve several of the embedded system design requirements previously mentioned. Specifically, in [10], SCE has been applied to the design of an ARM-based system utilizing both host-compiled and cycle accurate ARM simulation models for verification of the designed software and system performance.

Cycle accurate simulators emulate the hardware in full detail, giving time accurate results even before the physical model is available. However, the benefit of having high timing accuracy comes at the price of slow simulation runs. Much faster simulation results can be obtained by using host-compiled simulators which run at host machine speed. Nevertheless, they cannot check the functionality of the synthesized binary code. An intermediate approach is to use functional Instruction Set Simulators (ISS). Modern ISSs run at near host speed by doing dynamic translation of the target machine into the host architecture. ISSs can only provide approximate timing information, but should guarantee correct functionality.

One variant of ISS-based simulators are the so-called Virtual Platform Emulators. Virtual Platform Simulators extend the concept of ISSs to hardware peripherals besides the CPU. It also contains mechanisms for connecting the different elements in the designed platform. Virtual Platforms posses the main advantage of being able to run complete unmodified Operating Systems.

In our work, we extended the SCE framework to include a Virtual Platform simulator. This adds another option in the speed/accuracy space, between pin-accurate or transaction-level host-compiled and cycle-accurate simulators, allowing designers to quickly verify the target machine synthesized code in a homogeneous environment.

1.1 BACKGROUND

Current systems are continuously increasing in complexity. Multiple cores, connected through different busses and a diversified list of peripherals need to be integrated into a single testing and prototyping platform. Take into account, for instance, the simplified view of a smart-phone showed in Figure 1.1

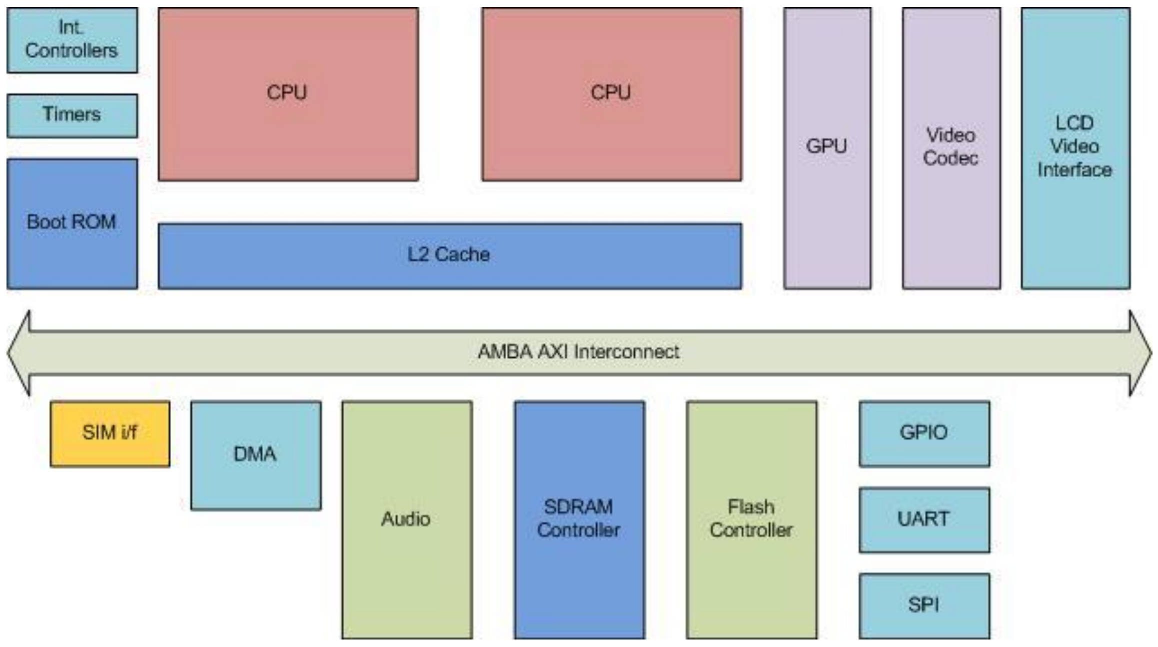


Figure 1.1: Multi-Processor System-on-Chip (MPSoC).

It contains multiple CPUs, hardware accelerators, different type of memories, and external interfaces. Designers would be interested not only in checking these components individually, but also as a whole in order to verify its interactions.

Also, the availability of different CPUs, busses and other hardware models are critical for a rapid exploration of the hardware space. Ideally, this should be performed automatically one day. Nevertheless, just the availability of different simulation models and the capability of quickly interchanging them manually help reduce considerably the time spent in taking design decisions.

1.1.1 System Level Design

Despite all the effort the EDA industry has invested in trying to reduce the productivity gap, the increasing complexity of the systems has made things worse. Current embedded systems not only have a variety of CPUs and peripherals interconnected but might also run several Operating Systems (OSs), which manage the different software parts. This results in the necessity of having parallel hardware and software design.

System-Level Design reduces the product design cycle significantly by allowing hardware and software engineers to work in parallel. This also helps detecting interface design bugs earlier.

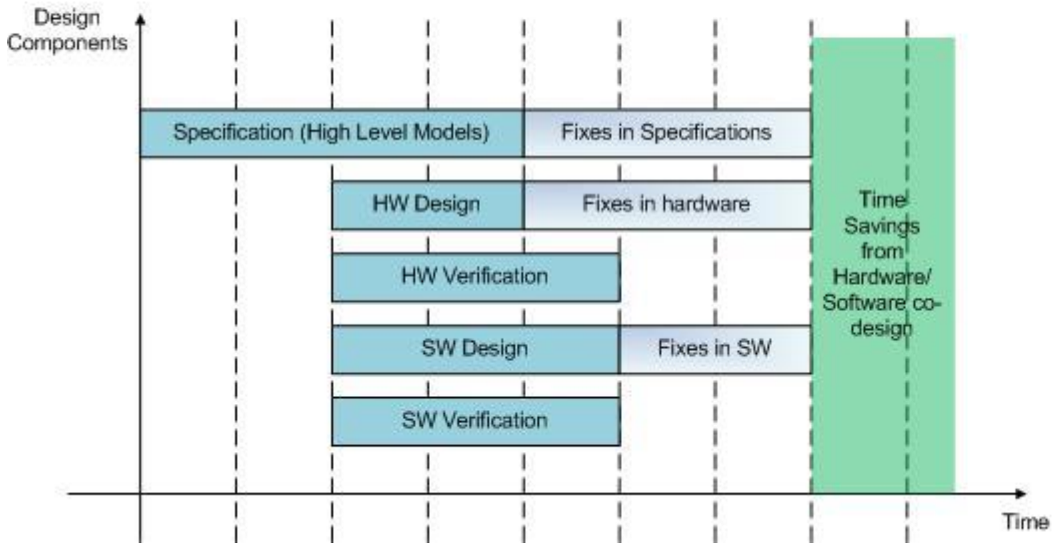


Figure 1.2: Hardware-software co-design product design cycle.

In SLD, through a set of tools or framework, designers can gradually refine the computational and communicational blocks. It is desirable that this refinement flow to be capable of representing all the system's modules at different levels of abstraction. Typical levels of abstraction used in modeling of embedded systems start at the requirement level, going down through the functional, structural, bus-functional, RTL (Register Transfer Level), and ending down at the Gate level model. This is called the Top down design.

A different approach is the so-called Platform-based design. In Platform-based design the system is built by merging together the individual base elements of the system, which are already defined in detail. For instance, HW/SW components are pre-defined and CPUs, busses and peripheral hardware selected. Only certain parameters, such as cache sizes or CPU frequency, are customizable. In this methodology, a higher productivity is obtained through design reuse. Although, being a straight forward design methodology, it limits the designer’s freedom during the design process.

Contrarily, the top-down design provides the possibility of exploring different HW platforms, without writing a single line of code or modeling new hardware, by making use of high abstraction representation of the systems modules.

1.1.2 System-Level Design Languages and Models

Abstract models of the system requirements help separate the specification from implementation, while the separation of computation and communication allows refining processing elements and communication mechanisms separately.

Ideally, a model of computation should be unbiased towards implementation either in hardware or software. Examples of models of computation are Synchronous Dataflow, State Machines, Kahn Process Networks, and Process State Machines [36].

While in models of communication, typically, Transaction Level or Pin Accurate Models are used. In Transaction Level Modeling (TLM), communication mechanisms are modeled as channels and interface functions handle transactions between computational blocks. TLM abstracts the details of the communication protocol and focus on the functionality of the data transfer. Contrarily, the Pin Accurate Modeling (PIM) is an accurate description of the communication protocol down to the pins and wires. All being said about models of computation and communication, a language is needed in order to express the system behavior in a concrete form. The language represents the model in a machine-readable fashion. Examples of languages used in industry and academia for representing the technical specifications of a system are: C, Java, VHDL [3], Verilog [4], SystemC [5], and SpecC [6].

It is not possible to model software entities with VHDL and Verilog since they are Hardware Description Languages (HDLs), which focus on cables, busses, pins, state changes, memories, etc. On the other hand, pure software languages such as C and Java, cannot model hardware details on timing and communication. Therefore, languages such as SystemC and SpecC are more suitable for Embedded Systems design since both of them can model Hardware and Software components.

A language may capture more than one model of computation and/or communication. Those languages that capture models of computation and communication and can express the specifications of a system at the highest level are called System-Level Design Languages (SLDLs).

An SLDL should also cover all concepts commonly found in Embedded Systems Design [7]: Behavioral and structural hierarchy that includes concurrency, synchronization, exception handling and timing. Besides, it should model state transitions and all these concepts should be organized orthogonally.

1.1.3 Electronic System-Level Design Frameworks

Unfortunately, even when there are many ways of representing a system at different levels of granularity, engineers usually need to go through the refinement steps (higher to lower abstraction levels) manually. Embedded Systems designers have to rely heavily on past experience or use a Platform-based design. Equally bad is the fact that different abstraction models and system's components need to be tested in different environments (e.g. Instruction Set Simulators, Network Simulators, Virtual Platforms, RTL simulators, etc.). Therefore an integrated development environment that combines all the tools is highly advisable.

A framework is the glue that connects seamlessly all tools in the SLD environment. Academia has been working on this issue for some years now, and EDA companies (after an unsuccessful first push) have recently been trying to fill up this hole.

Examples of academic tools are SCE, Daedalus [31], Metropolis [32] and SystemCoDesigner [33], while commercial examples are Synopsis' Synphony [34] and Mentor Graphics' Catapult-C [35].

Some of the key features that a SLD framework should have are:

1. To accept one or more models of computation and communication of the system's specifications.
2. Fast simulation (Host-compiled simulators) and profiling tools.
3. Different processor models, with different granularity (e.g. cycle-accurate, instruction-accurate), that can be used to simulate and verify the embedded software being synthesized.
4. The embedded software synthesizer tool targeted for the selected OS.
5. Hardware synthesis tool.

1.2 THE SYSTEM-ON-CHIP ENVIRONMENT (SCE)

The System-On-Chip Environment (SCE) [9], developed at UC-Irvine, is a framework that builds on the SpecC modeling language with its compiler and simulator, and lets designers automatically refine system specifications into lower levels of abstraction. In the next two sub-sections we will provide more details on the SpecC SLDL and on how the SLD flow is managed in SCE.

1.2.1 The SpecC SLDL

SpecC is a superset of ANSI-C where every C program becomes a SpecC program. While in an ANSI-C program each program is a set of functions, in SpecC programs become a set of "Behaviors, channels, and interfaces". The concept of a "Behavior" is similar to that of an entity in VHDL.

Behaviors connect to each other through channels and variables, while interfaces are the input/output methods of these behaviors.

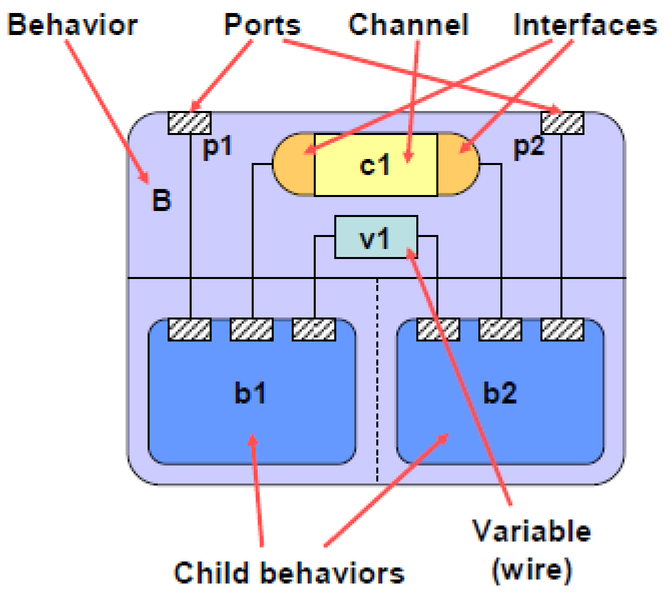


Figure 1.3: SpecC Behaviors, channels and interfaces [2].

SpecC supports all ANSI-C data types, plus has explicit support of truth values and bit vectors of arbitrary length. Furthermore, it has support for synchronization (events) and RTL concepts (such as buffered and signal).

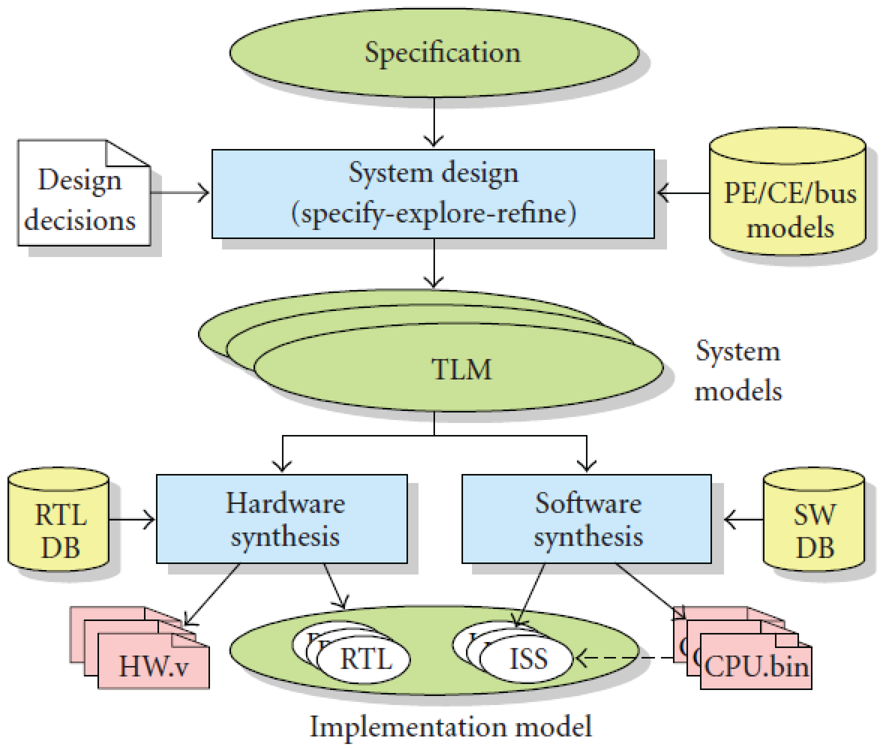
SpecC clearly defines keywords to support behavioral hierarchy, such as sequential, finite-state-machine (FSM), concurrent, and pipelined execution.

Communication between behaviors can be performed through shared variables, virtual, and hierarchical channels. The SpecC standard channel library includes support for semaphores, mutexes, critical sections, barriers, tokens, handshakes and double handshakes.

Finally, SpecC supports explicit timing by use of the waitfor <delay> statement. Time is independent from the host machine speed and is managed by the simulation kernel.

1.2.2 SCE Design Flow

In order to give a better picture on how our contribution adds up to the SCE



framework, we shall briefly discuss how the SCE design flow works.

Figure 1.4: SCE design flow [9].

In SCE, designers start by entering the golden specification of the system and then go through the different refinement processes by making design decisions on each step through a Graphical User Interface (GUI). Refinement is referred to as the course of moving from one abstraction level to the next one. Figure 1.4 shows a simplified diagram of the SCE design flow.

After entering the specification of the system in the form of SpecC behaviors and channels, designers map these behaviors and communication channels to Processing Elements (PEs), Communicational Elements (CEs) and busses. Based on the profiling results, designers can mix and change different models from the SCE database in order to quickly explore the design space; and by relying on the user's design decisions, SCE automatically generates models that have an increasing amount of implementation details [9].

The result of the system design phase is a Transaction Level Model (TLM), which is a time accurate representation of the system architecture. Also, a Pin Accurate Model (PAM) can be generated at this stage. The PAM represents detailed communication protocol information down to the wires and pins.

From here, the HW/SW synthesis generates the RTL and instruction set specific code implementation models and the SW binary images can be verified against an instruction set simulator (ISS).

1.3 Problem Statement and Approach

Currently, the SCE framework is limited to verifying the automatically generated software binaries against cycle-accurate CPU simulators. It guarantees behavioral correctness and provides accurate timing information, but at the cost of long simulation times. Also available in SCE is the ability to check the different system blocks with host-compiled simulation. Although this provides fast simulation results, it cannot guarantee correct target execution of the final binaries.

Therefore, we wanted to extend the SCE framework capabilities for fast hardware exploration and verification of the synthesized software through the addition of an interface to a functional Instruction-Set Simulator (ISS). Particularly, we were interested in adding the interface to an ISS-centric Virtual Platform model [37].

For that matter, we proceeded as follows: First, we studied the related work on integration of virtual platform emulators and system-level design tools. This resulted in two major choices for Virtual Platform emulators to be integrated into SCE: QEMU [12] and OVP [16]. Both have been integrated into SystemC in the past, which was a pretty good indicator that they could also be integrated into SpecC and its corresponding framework (SCE).

After choosing one of these platform emulators, we proceeded to relate what was done previously by others to the SCE framework and SpecC Language; identifying benefits and potential pitfalls. This examination resulted in the architecture of our SpecC wrapper.

Next, we adapted our test application model to the interoperation limitations (different cross-compilers, different supported RTOS, etc.).

Following, we verified that our modified test application was still working with a cycle accurate model, which was previously integrated into the SCE framework. The timing results from the cycle accurate model became our golden reference.

We then proceeded to integrate our wrapper into the SCE framework and verified that the results were functional correct.

Finally, we compared the golden reference results to ours, and analyzed the results based on execution time and accuracy of the results.

1.4 OUTLINE OF THE REPORT

The rest of this report is organized as follows: In Chapter 2 we describe advantages and disadvantages of both OVP and QEMU, along with a description of previous integrations with SystemC. At the end of the chapter we provide a comparison table according to our observations.

In Chapter 3 we describe the design of our SpecC/OVP Wrapper. We also provide a short description of how the wrapper integrates into the SCE hardware exploration and design flow.

Finally, we have included our experiment results in Chapter 4, while Chapter 5 holds our conclusions and future work.

CHAPTER 2

Related Work

In order to understand program behavior and finding bottlenecks in early stages of the product cycle, embedded systems design has relied heavily on different types of simulators. Traditionally, simulations are performed in cycle accurate models, which take weeks to run completely for modern benchmarks like the SPEC suites [18].

Schirner et. al. [10] used SWARM [11] as the hardware emulator for simulating cycle-accurate code inside the SCE framework. Although very useful for verifying system requirements, it is constrained to a single architecture. Besides, simulation time is still slow for running unmodified larger software applications.

Several researchers have proposed to use hardware accelerated frameworks [22] to improve simulation time compared to software cycle accurate simulators. However, the usage of an FPGA and synthesizable version of the hardware in early stages of the development cycle might be too expensive. Another method for trying to reduce the simulation time of benchmarks is to install simulation points [19]. But that requires a deeper analysis of the code, and again limits the testing and profiling to only certain parts of the software.

There is obviously a tradeoff between having faster simulation models and more precise ones. As discussed before, another approach to increase productivity and reduce simulation time is to integrate system-level design models with host-compiled simulators. The SCE framework already comes with its host-compiled simulation, which provides a fast verification method of the functionality and timing of the design. Nevertheless, the behavior of the simulated system will differ in important matters as size of memory,

memory protection and other issues that can make code that works well in simulation break on the real target. Equally important is the fact that host-compiled binaries will not run in the target architecture.

The scheme we focus in our report is in the verification of the target synthesized code through the use of Instruction Set Simulators (ISS). Software engineers have been using ISS for some time, not only for software design, but also as PC virtualization.

2.1 INSTRUCTION SET SIMULATORS

ISSs are required to run several orders of magnitude faster than fully cycle accurate microarchitecture simulators. Instruction Set Simulators can be divided in:

- Interpretive
- Static compiled
- Dynamic translation

Interpretive ISS are flexible but slow. Instructions are fetched, decoded and executed at run time. This can provide detailed accuracy, but makes the simulation slow.

Static compiled ISSs do compile-time decoding of the target machine code into the host machine, which gives a better simulation performance. Nevertheless, the complete program code needs to be available at compile time, limiting flexibility.

Dynamically translating ISSs are similar to compiled ISSs, but give more flexibility to handle dynamic situations, such as self-modifying code. They move the translation step into the simulation run-time. In order to gain speed and avoid unnecessary retranslation, modern dynamic ISSs keep the translated code in a translation cache. There are other variations to dynamic translation that try to increase the simulation performance by using various different techniques [38] [39] and inspecting all of them is beyond the scope of this report.

ISS may be encapsulated into virtual platform emulators, such as QEMU [12] and OVP [16]. Virtual Platform emulators not only consider code translation, but also the modeling of peripherals and other ancillary features of a microprocessor, such as MMUs, timers, etc. As such, Virtual Platform emulators allow executing unmodified guest Operating Systems on them.

2.2 Virtual Platforms

Virtual Platforms, also known as Virtual Machines, are available both freely and commercially. Sometimes companies take a dual approach, distributing free licenses for personal and/or educational usage, or making a limited version of the emulator freely available. This is the case for VirtualBox [23], VMware [24], and OVP. On the other hand, QEMU is distributed freely under the GNU GPL.

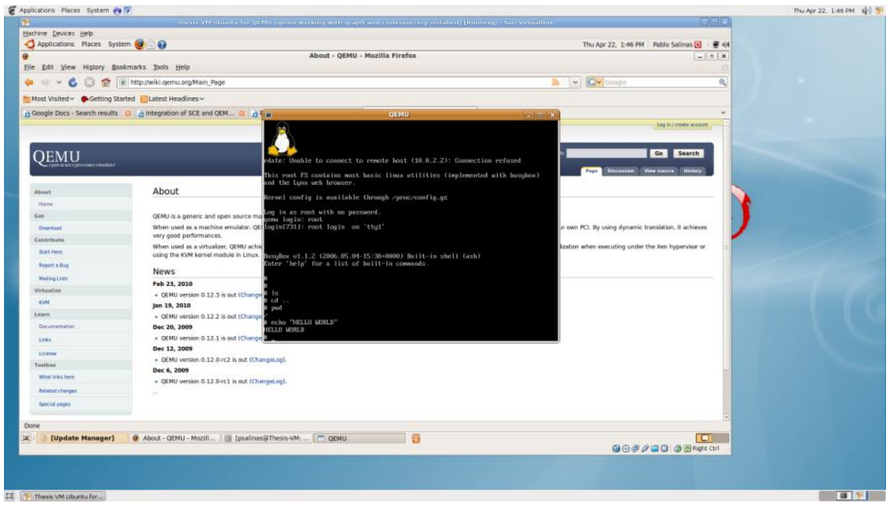
2.2.1 QEMU

QEMU is a platform emulator that uses dynamic translation to achieve faster emulation speed. It has two operating modes:

a. Full system emulation: A full system is emulated, which includes one or more processors and various peripherals.

b. User mode emulation: In this mode, processes compiled for one CPU can be run on another CPU. It is intended just to run simple applications, mostly with no libraries and is currently limited to Linux and Darwin/Mac OS X processes.

Figure 2.1 shows a QEMU emulation of an ARM running Linux inside a VirtualBox in a Debian i7 CPU.



```
rdate: Unable to connect to remote host (10.0.2.2): Connection refused

This root FS contains most basic linux utilities (implemented with busybox)
and the Lynx web browser.

Kernel config is available through /proc/config.gz

Log in as root with no password.
qemu login: root
login[731]: root login  on `tty1'


BusyBox v1.1.2 (2006.05.04-15:30+0000) Built-in shell (ash)
Enter 'help' for a list of built-in commands.

#
#
# ls
# cd ..
# pwd
/
# echo "HELLO WORLD"
HELLO WORLD
#
```

Figure 2.1: QEMU emulating ARM processor; running inside VirtualBox.

QEMU also provides support for virtualization in cases where the host and target CPU are the same, or through the use of KVM (Kernel-based Virtual Machine) [25].

QEMU portable dynamic translation

To allow easy porting of the ISS to new host or target architectures, QEMU first decompiles the simulation binary into fewer simpler instructions called micro operations. These micro operations are written as C code that gets compiled by GCC to an object file. A compile-time tool called dyngen generates a dynamic code generator based on the input object file. During simulation time, the dynamic code generator is then invoked and generates a complete host function that concatenates micro operations.

QEMU achieves fast emulation by doing target to host translation in blocks. Translated Blocks consist of sequences of target code up to the next jump or instruction modifying the CPU state (PC, registers, etc.) in a way that cannot be deduced at translation time. Also, a small cache holds the most recently TBs to avoid unnecessary code re-translation.

Other details regarding the QEMU architecture

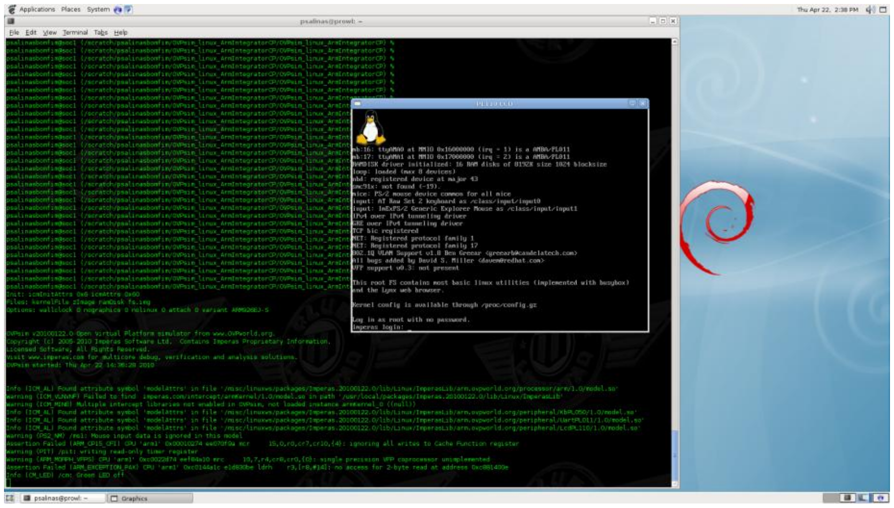
Similar to other emulators, the communication between different emulated devices is done via callback functions registered for each memory region of the system bus. QEMU supports precise exceptions by always being able to retrieve the target CPU state at the time the exception has occurred. On the other hand, for hardware interrupts to be detected, the user (or emulated hardware designer) needs to call a specific API function to indicate that an interrupt is pending.

Overall QEMU is a fast, freely available simulator with support for many Operating Systems and target architectures. Nevertheless, it has a few problems, such as the fact of having no real concept of simulation time and poorly documented APIs.

2.2.2 OVP

In March, 2008, Imperas (a company based in the U.K.) released its Open Virtual Platforms to the EDA community. All OVP models were released as Open Source under the Apache 2.0 license. Also, the OVP simulation engine and the OVP modeling APIs are free for non-commercial use.

Although the fact that a for-profit company is the main supporter of the OVP technology might be subject of concern, they claim that their strategy is to "build its business around software verification technologies, multi-core development, and other complementary solutions" [26].



```
psalinasbomfim@soc1 (/scratch/psalinasbomfim/OVPsim_linux_ArmIntegratorCP/OVPsim_linux_ArmIntegratorCP) %
psalinasbomfim@soc1 (/scratch/psalinasbomfim/OVPsim_linux_ArmIntegratorCP/OVPsim_linux_ArmIntegratorCP) %
psalinasbomfim@soc1 (/scratch/psalinasbomfim/OVPsim_linux_ArmIntegratorCP/OVPsim_linux_ArmIntegratorCP) %
psalinasbomfim@soc1 (/scratch/psalinasbomfim/OVPsim_linux_ArmIntegratorCP/OVPsim_linux_ArmIntegratorCP) %
psalinasbomfim@soc1 (/scratch/psalinasbomfim/OVPsim_linux_ArmIntegratorCP/OVPsim_linux_ArmIntegratorCP) %
psalinasbomfim@soc1 (/scratch/psalinasbomfim/OVPsim_linux_ArmIntegratorCP/OVPsim_linux_ArmIntegratorCP) %
psalinasbomfim@soc1 (/scratch/psalinasbomfim/OVPsim_linux_ArmIntegratorCP/OVPsim_linux_ArmIntegratorCP) %
psalinasbomfim@soc1 (/scratch/psalinasbomfim/OVPsim_linux_ArmIntegratorCP/OVPsim_linux_ArmInt
psalinasbomfim@soc1 (/scratch/psalinasbomfim/OVPsim_linux_ArmIntegratorCP/OVPsim_linux_ArmInt
psalinasbomfim@soc1 (/scratch/psalinasbomfim/OVPsim_linux_ArmIntegratorCP/OVPsim_linux_ArmInt
psalinasbomfim@soc1 (/scratch/psalinasbomfim/OVPsim_linux_ArmIntegratorCP/OVPsim_linux_ArmInt
psalinasbomfim@soc1 (/scratch/psalinasbomfim/OVPsim_linux_ArmIntegratorCP/OVPsim_linux_ArmInt
psalinasbomfim@soc1 (/scratch/psalinasbomfim/OVPsim_linux_ArmIntegratorCP/OVPsim_linux_ArmInt
psalinasbomfim@soc1 (/scratch/psalinasbomfim/OVPsim_linux_ArmIntegratorCP/OVPsim_linux_ArmInt
psalinasbomfim@soc1 (/scratch/psalinasbomfim/OVPsim_linux_ArmIntegratorCP/OVPsim_linux_ArmInt
psalinasbomfim@soc1 (/scratch/psalinasbomfim/OVPsim_linux_ArmIntegratorCP/OVPsim_linux_ArmInt
psalinasbomfim@soc1 (/scratch/psalinasbomfim/OVPsim_linux_ArmIntegratorCP/OVPsim_linux_ArmInt
psalinasbomfim@soc1 (/scratch/psalinasbomfim/OVPsim_linux_ArmIntegratorCP/OVPsim_linux_ArmInt
psalinasbomfim@soc1 (/scratch/psalinasbomfim/OVPsim_linux_ArmIntegratorCP/OVPsim_linux_ArmInt
psalinasbomfim@soc1 (/scratch/psalinasbomfim/OVPsim_linux_ArmIntegratorCP/OVPsim_linux_ArmInt
psalinasbomfim@soc1 (/scratch/psalinasbomfim/OVPsim_linux_ArmIntegratorCP/OVPsim_linux_ArmInt
psalinasbomfim@soc1 (/scratch/psalinasbomfim/OVPsim_linux_ArmIntegratorCP/OVPsim_linux_ArmInt
psalinasbomfim@soc1 (/scratch/psalinasbomfim/OVPsim_linux_ArmIntegratorCP/OVPsim_linux_ArmInt
psalinasbomfim@soc1 (/scratch/psalinasbomfim/OVPsim_linux_ArmIntegratorCP/OVPsim_linux_ArmInt
psalinasbomfim@soc1 (/scratch/psalinasbomfim/OVPsim_linux_ArmIntegratorCP/OVPsim_linux_ArmInt
psalinasbomfim@soc1 (/scratch/psalinasbomfim/OVPsim_linux_ArmIntegratorCP/OVPsim_linux_ArmInt
psalinasbomfim@soc1 (/scratch/psalinasbomfim/OVPsim_linux_ArmIntegratorCP/OVPsim_linux_ArmInt
psalinasbomfim@soc1 (/scratch/psalinasbomfim/OVPsim_linux_ArmIntegratorCP/OVPsim_linux_ArmInt
psalinasbomfim@soc1 (/scratch/psalinasbomfim/OVPsim_linux_ArmIntegratorCP/OVPsim_linux_ArmInt
psalinasbomfim@soc1 (/scratch/psalinasbomfim/OVPsim_linux_ArmIntegratorCP/OVPsim_linux_ArmInt
psalinasbomfim@soc1 (/scratch/psalinasbomfim/OVPsim_linux_ArmIntegratorCP/OVPsim_linux_ArmInt
psalinasbomfim@soc1 (/scratch/psalinasbomfim/OVPsim_linux_ArmIntegratorCP/OVPsim_linux_ArmInt
psalinasbomfim@soc1 (/scratch/psalinasbomfim/OVPsim_linux_ArmIntegratorCP/OVPsim_linux_ArmInt
psalinasbomfim@soc1 (/scratch/psalinasbomfim/OVPsim_linux_ArmIntegratorCP/OVPsim_linux_ArmInt
psalinasbomfim@soc1 (/scratch/psalinasbomfim/OVPsim_linux_ArmIntegratorCP/OVPsim_linux_ArmInt
psalinasbomfim@soc1 (/scratch/psalinasbomfim/OVPsim_linux_ArmIntegratorCP/OVPsim_linux_ArmInt
psalinasbomfim@soc1 (/scratch/psalinasbomfim/OVPsim_linux_ArmIntegratorCP/OVPsim_linux_ArmInt
psalinasbomfim@soc1 (/scratch/psalinasbomfim/OVPsim_linux_ArmIntegratorCP/OVPsim_linux_ArmInt
psalinasbomfim@soc1 (/scratch/psalinasbomfim/OVPsim_linux_ArmIntegratorCP/OVPsim_linux_ArmInt
psalinasbomfim@soc1 (/scratch/psalinasbomfim/OVPsim_linux_ArmIntegratorCP/OVPsim_linux_ArmInt
psalinasbomfim@soc1 (/scratch/psalinasbomfim/OVPsim_linux_ArmIntegratorCP/OVPsim_linux_ArmInt
psalinasbomfim@soc1 (/scratch/psalinasbomfim/OVPsim_linux_ArmIntegratorCP/OVPsim_linux_ArmInt
Init: icmInitAttrs 0x6 icmAttrs 0x60
Files: kernelFile zImage ramDisk fs.img
Options: wallclock 0 nographics 0 nolinux 0 attach 0 variant ARM926EJ-S


OVPsim v20100122.0 Open Virtual Platform simulator from www.OVPworld.org.
Copyright (c) 2005-2010 Imperas Software Ltd.  Contains Imperas Proprietary Information.
Licensed Software, All Rights Reserved.
Visit www.imperas.com for multicore debug, verification and analysis solutions.
OVPsim started: Thu Apr 22 14:36:28 2010


Info (ICM_AL) Found attribute symbol 'modelAttrs' in file '/misc/linuxws/packages/Imperas.20100122.0/lib/Linux/ImperasLib/arm.ovpworld.org/processor/arm/1.0/model.so'
Warning (ICM_VLNVNF) Failed to find  imperas.com/intercept/armKernel/1.0/model.so in path '/usr/local/packages/Imperas.20100122.0/lib/Linux/ImperasLib'
Warning (ICM_MINE) Multiple intercept libraries not enabled in OVPsim, not loaded instance armKernel_0 ((null))
Info (ICM_AL) Found attribute symbol 'modelAttrs' in file '/misc/linuxws/packages/Imperas.20100122.0/lib/Linux/ImperasLib/arm.ovpworld.org/peripheral/KbPL050/1.0/model.so'
Info (ICM_AL) Found attribute symbol 'modelAttrs' in file '/misc/linuxws/packages/Imperas.20100122.0/lib/Linux/ImperasLib/arm.ovpworld.org/peripheral/UartPL011/1.0/model.so'
Info (ICM_AL) Found attribute symbol 'modelAttrs' in file '/misc/linuxws/packages/Imperas.20100122.0/lib/Linux/ImperasLib/arm.ovpworld.org/peripheral/LcdPL110/1.0/model.so'
Warning (PS2_NM) /ms1: Mouse input data is ignored in this model
Assertion Failed (ARM_CP15_CPI) CPU 'arm1' 0x00010274 ee070f9a mcr     15,0,r0,cr7,cr10,{4}: ignoring all writes to Cache Function register
Warning (PIT) /pit: writing read-only timer register
Warning (ARM_MORPH_VFPS) CPU 'arm1' 0xc0022d74 eef84a10 mrc     10,7,r4,cr8,cr0,{0}: single precision VFP coprocessor unimplemented
Assertion Failed (ARM_EXCEPTION_PAX) CPU 'arm1' 0xc0144a1c e1d830be ldrh    r3,[r8,#14]: no access for 2-byte read at address 0xc881400e
Info (CM_LED) /cm: Green LED off
```

```
mb:16: ttyAMA0 at MMIO 0x16000000 (irq = 1) is a AMBA/PL011
mb:17: ttyAMA1 at MMIO 0x17000000 (irq = 2) is a AMBA/PL011
RAMDISK driver initialized: 16 RAM disks of 8192K size 1024 blocksize
loop: loaded (max 8 devices)
nbd: registered device at major 43
smc91x: not found (-19).
mice: PS/2 mouse device common for all mice
input: AT Raw Set 2 keyboard as /class/input/input0
input: ImExPS/2 Generic Explorer Mouse as /class/input/input1
IPv4 over IPv4 tunneling driver
GRE over IPv4 tunneling driver
TCP bic registered
NET: Registered protocol family 1
NET: Registered protocol family 17
802.1Q VLAN Support v1.8 Ben Greear <greearb@candelatech.com>
All bugs added by David S. Miller <davem@redhat.com>
VFP support v0.3: not present

This root FS contains most basic linux utilities (implemented with busybox)
and the lynx web browser.

Kernel config is available through /proc/config.gz

Log in as root with no password.
imperas login:
```

Figure 2.2: OVP emulating an ARM Integrator platform running Linux.

OVP is made up of four C interfaces: ICM for creating platforms, VMI for processors, BHM for behavioral blocks and PPM for peripherals. Similar to QEMU, OVP uses code morphing and a just-in-time (JIT) compiler. As with QEMU, target CPU operations are mapped into a set of optimized opcodes before being translated into the host CPU operations [27].

Virtualization is also provided through the VMI interface, which allows direct execution on the host using the included standards libraries.

The four APIs released by OVP allow embedded system designers to model complete platforms. These APIs are well documented and many models and platforms examples are given.

In OVP, each peripheral model runs in a separate virtual machine with its own address space. This is called the Peripheral Simulation Engine or PSE. While peripheral code is running, time is not advanced in the simulator. Nevertheless, simulated delays, events and thread synchronization can be achieved by using the BHM (Behavioral Hardware Modeling) API.

2.2.3 SLDL Integration

Wang et. al. [13], and Monton et. al. [14] have integrated QEMU with SystemC. Monton's work is currently part of the Greensocs [15] projects. Greensocs is a U.K. non-profit that makes "IP available publicly on the GreenSocs web site". Basically, any IP downloaded from Greensocs is under the GNU GPL, unless another contract agreement is established between the user and Greensocs.

We have focused most of our research on the work by Monton and colleagues at Greensocs, since that code is freely available through the GreenSocs website and forum. Two versions of the QEMU-System integration exist: One consists of a SystemC bridge that connects to QEMU through the PCI (or AMBA) interface, which is provided as part of the QEMU API. The other consists of a SystemC wrapper. The wrapper contains an SC_THREAD that runs a modified QEMU main loop and calls SystemC wait() statements for synchronization of the QEMU and SystemC clocks.

By contrast, Imperas has already created a C++ wrapper for integration of their OVP models with SystemC TLM-2.0 modules. This wrapper was required for both interconnection of two different simulators and due to the fact that OVP's original API is written in C, while SystemC is a C++ library.

Since in our case, we were using SpecC, no language conversion is required for interfacing to OVP's APIs. Nevertheless, an API language conversion is required for interfacing with QEMU's APIs (from C++ to C).

2.3 QEMU VS. OVP

Taking into account what was expressed in chapters 2.2.1 and 2.2.2, information collected from the web and our personal experiences, we have identified some of QEMU and OVP key features. A table with our findings is provided in the Appendix of this report.

Nevertheless, our analysis needs to go beyond identifying key features of both emulators. We also need to be the least intrusive to the Virtual Platform emulator as possible and take into account the difficulty of integrating it into a System-Level Design Tool. Therefore, Table 1 correlates the integration of QEMU and OVP to SystemC in order to give a better picture of what to expect when integrating one of them into our framework.

	QEMU	**OVP**
Integration into SLDL required modification to the simulation source code?	Yes.	No. OVP sources already come with required SystemC wrappers and APIs for running/stopping the emulator
Simulation is driven by?	QEMU.	SystemC.
Simulators and processes communicate through?	A registered QEMU device (sc_link.c) that registers itself (the callback functions) as system bus device.	Registered callback functions.
TLM supported?	GreenSocs Generic Protocol Socket, which claims to be TLM2.0 compliant.	OSCI TLM2.0 is supported.

Time	No concept of time.	Behavioral approximate concept of time.
Documentation of APIs	Poorly documented.	Well documented which includes examples and with their corresponding descriptions.
Support	Forums.	Forum backed by Imperas.

Table 1: Comparison of integration of QEMU and OVP into SystemC.

QEMU's main advantage is the fact that it is a free open source project with a large database of ported target architectures and platforms. Nevertheless, its lack of documentation on the APIs or standardized methodology for creating new models and platforms makes it really hard to implement our wrapper.

Also, the fact that it would require modification of its source code would make it difficult to maintain compatibility with newer versions of QEMU. Just making the QEMU-SystemC wrapper provided by Greensocs to work was very time consuming. And we also had the experience that since their wrapper is still a work in progress, from one checkout to the next one, the Greensocs platform stopped working.

Contrarily, OVP offers a wealth of documentation on the APIs. It also includes many examples and, from our experience, the forum for Q&A is well maintained by Imperas' engineers (as by other contributors).

Therefore, we opted for integrating OVP into the SCE platform. In the next chapter, we will discuss the details of this work.

CHAPTER 3

SCE/OVP Integration

From our knowledge of the SCE framework and our study on previous integration between OVP and SystemC, we could deduct certain relevant points to be considered.

First, in order to closely match our host-compiled simulation to the synthesized software verification, we should choose a CPU model, namely an ARM7 processor, that was already included in our abstract model database.

Secondly, we need to develop a SpecC wrapper for inclusion of the ARM7 OVP ISS in the SCE component database. In the process, we need to decide how to perform the synchronization of both simulation environments. By inspecting the OVP APIs, we can build the interface, which should make use of any communication mechanisms available.

Lastly, we need to take into account the targeting of the wrapped OVP simulation model by the SCE software synthesis and target binary generation backend. If tasks mapped to the CPU are scheduled by an RTOS, we need to have the RTOS ported and tested against the OVP CPU model. Likewise, we have to ensure that our synthesized embedded software, including all drivers and interrupt handlers properly integrates into the SpecC-based OVP platform model.

3.1 WRAPPER

The SpecC/OVP wrapper is the module that connects the simulator kernel with the Virtual Platform, and allows them to communicate in a synchronized fashion.

Although a full platform could potentially live inside the OVP environment, we decided to focus on using the wrapper for a single CPU for the time being. Besides, it was our intention to primarily use the OVP platform as a fast ISS of CPU models, and leave the peripherals and bus modeling to be performed in SpecC, integrated through SCE. Particularly, we have chosen to use the ARM7TDMI CPU model as our ISS, since we already had a behavioral description of it in SpecC. This helps our wrapper to be easily integrated into the SCE refinement process. Within the context of the SCE flow, the SpecC/OVP wrapper is then selected at the last stage of verifying the synthesized binaries (instead of using the SWARM cycle accurate simulator as the verification platform).

In order to integrate our wrapper into the SCE processor database, we combined it with the necessary programmable interrupt controller (PIC) and timer models that we obtained from [10]. Connections between the CPU running inside the OVP wrapper and the outside world are done through an AMBA AHB bus model as shown in Figure 3.1. The CPU can be interrupted through the IRQ and FIQ connections coming from the Interrupt Controller. While any data communication from and to the CPU is performed through the AMBA bus.

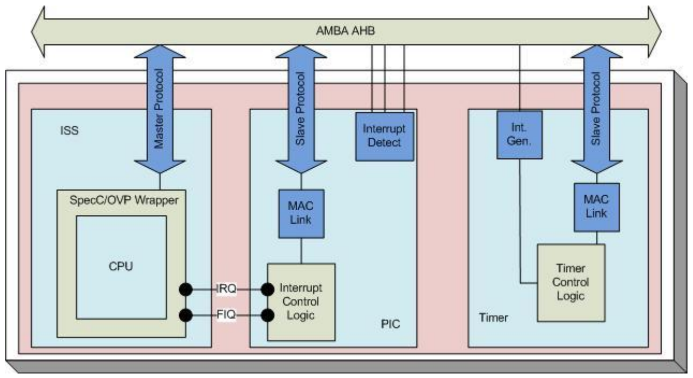


Figure 3.1: Connections between the wrapper and outside SpecC behaviors.

3.1.1 Interface

The OVP API [28] provides functions for controlling the execution of the OVP simulator and for interfacing with their CPU models:

- *icmInit* is used for initializing the simulation environment. It lets the programmer indicate certain aspects of simulation, as whether to produce trace information or how a debugger should be connected to the simulator if required.
- *icmNewProcessor* creates a new processor instance and allows indicating settings as: CPU name, type, and address bits.
- *icmLoadProcessorMemory* loads an object file into the processor memory. For the time of this report OVP supported only ELF binaries, while COFF files were still not supported.
- *icmSimulate* lets the indicated processor run up to the selected number of instructions.
- *icmNewNet*, *icmConnectProcessorNet* and *icmWriteNet* allow to create an interrupt signal, connect it to the selected CPU, and to write high or low values to it.
- *icmMapExternalMemory* specifies that accesses to certain memory ranges should generate calls to the write and read callback functions.

With the above functionalities at our disposal, we designed the SpecC/OVP wrapper algorithm as showed in Figure 3.2.

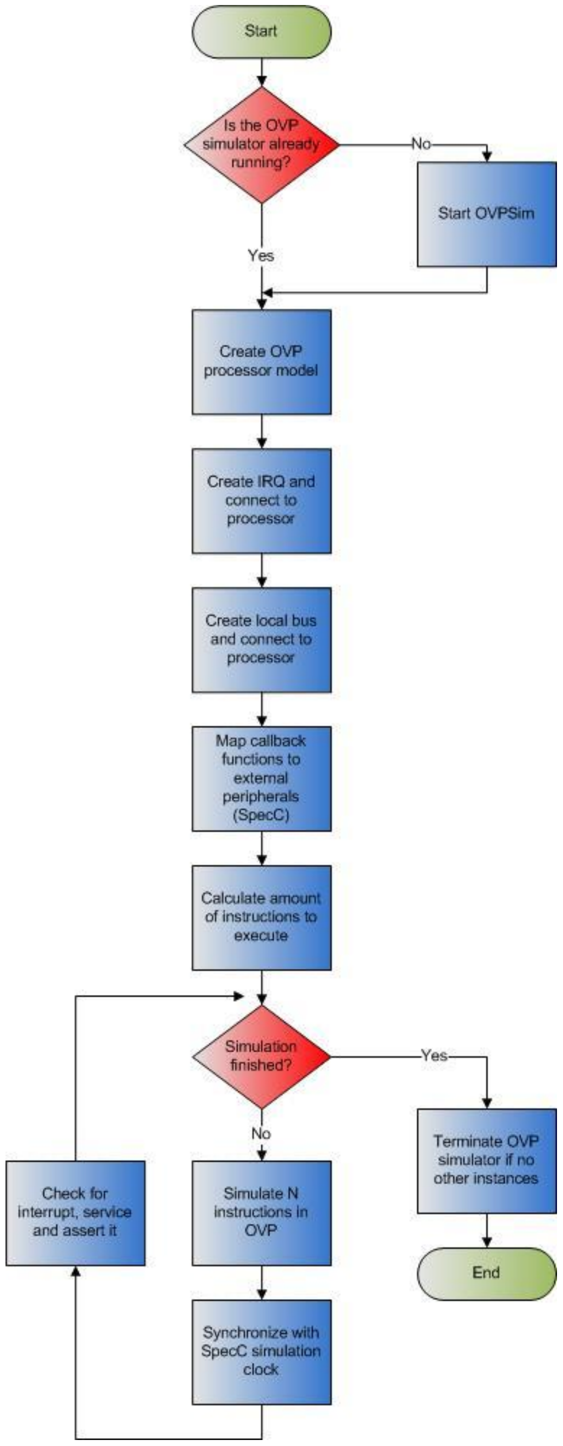

Start
Is the OVP simulator already running?
No
Start OVPSim
Yes
Create OVP processor model
Create IRQ and connect to processor
Create local bus and connect to processor
Map callback functions to external peripherals (SpecC)
Calculate amount of instructions to execute
Simulation finished?
Yes
Terminate OVP simulator if no other instances
End
No
Simulate N instructions in OVP
Synchronize with SpecC simulation clock
Check for interrupt, service and assert it

Figure 3.2: SpecC/OVP wrapper algorithm.

Whenever there is a read or write access from the ARM software to an externally mapped bus address, the registered callback functions within the wrapper are executed. These callback functions then call the SpecC AMBA interfaces for reading from or writing to the selected address. Furthermore, in the case of interrupts, interrupt signals are received from the PIC-connected interrupt lines. On each loop of the simulation, interrupts are checked, signaled to OVP and asserted or "deasserted".

3.1.2 Synchronization

The synchronization of the OVP and the SpecC simulator time is done as follows: A quantum time is set at the beginning of the wrapper invocation. During every loop of the SpecC/OVP wrapper execution (See Figure 3.2), the number of instructions that could be executed in quantum time (according to the CPU nominal performance) is run in the OVP simulator. Then, at the end of each iteration of the loop, the SpecC *waitfor*(quantum) advances the simulation time by the value of the quantum. This is shown graphically in Figure 3.3.

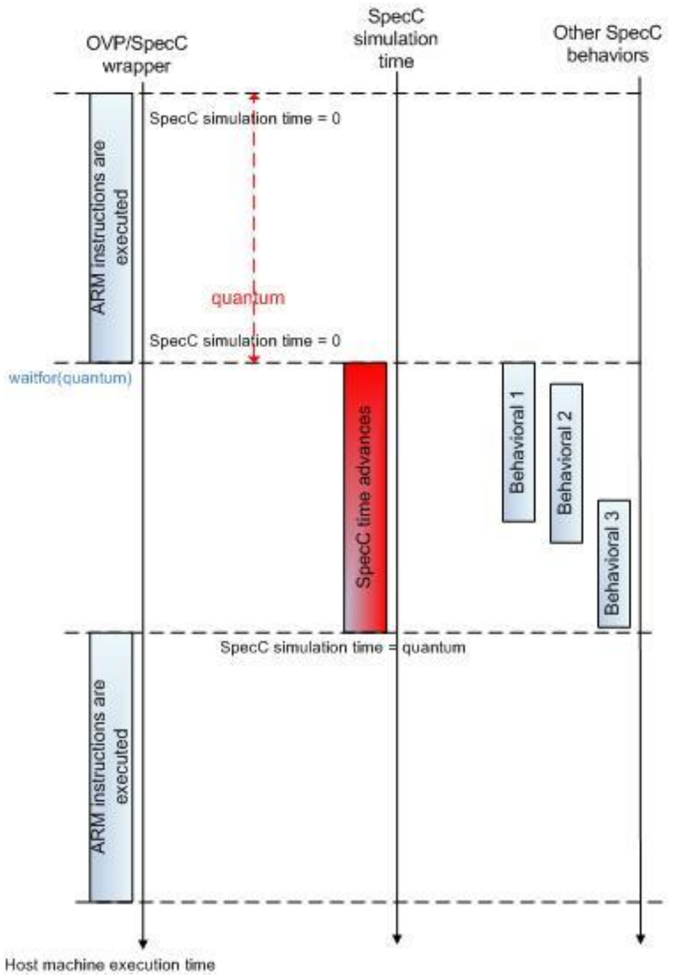

OVP/SpecC wrapper
SpecC simulation time
Other SpecC behaviors
SpecC simulation time = 0
ARM instructions are executed
quantum
SpecC simulation time = 0
waitfor(quantum)
SpecC time advances
Behavioral 1
Behavioral 2
Behavioral 3
SpecC simulation time = quantum
ARM instructions are executed
Host machine execution time

Figure 3.3: OVP simulator and SpecC simulation kernel time synchronization.

In SpecC, the simulation time only advances once an explicit call to *waitfor* is executed. Otherwise, everything inside that behavior runs in zero simulated time. This helps reduce the overhead of context switching but introduces a timing error. Specially, if the quantum value is too large when compared to the elapsed time between bus accesses and interrupts. Consider for instance the situation presented in Figure 3.4.

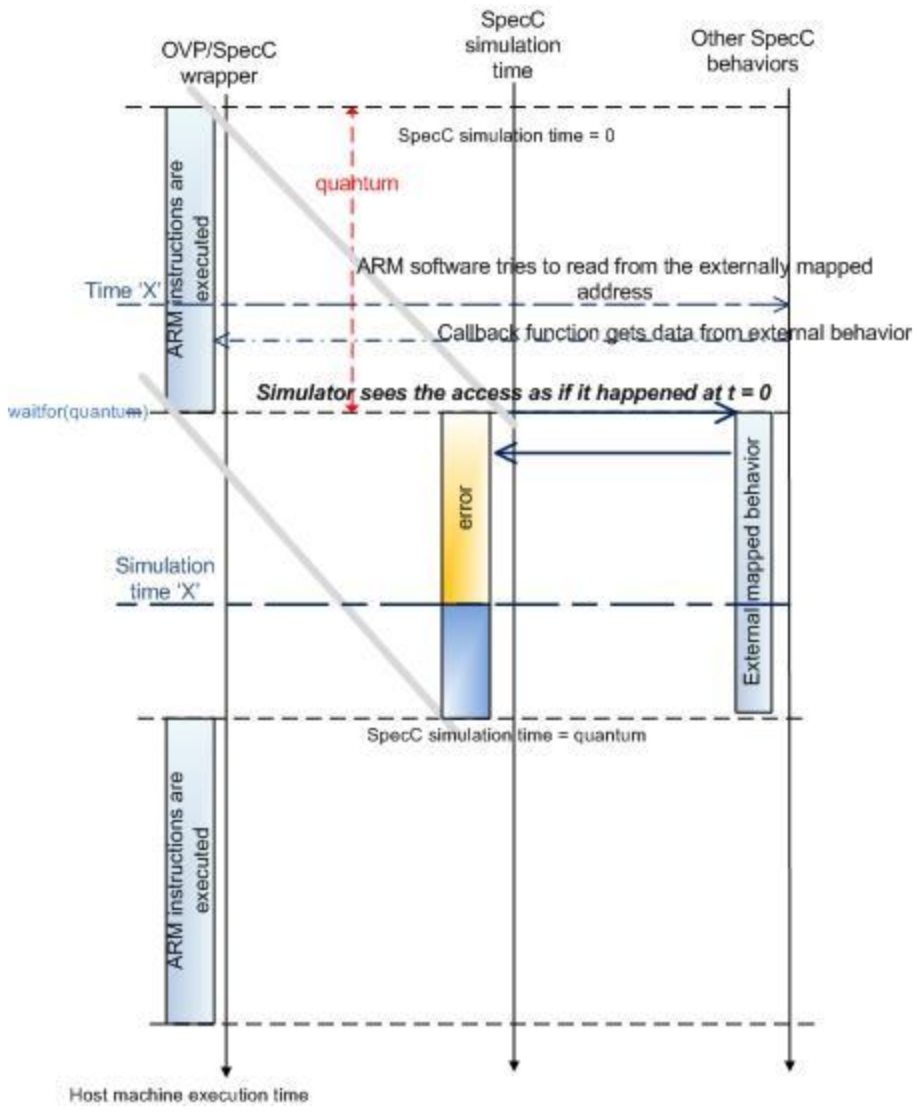

OVP/SpecC wrapper
SpecC simulation time
Other SpecC behaviors
SpecC simulation time = 0
quantum
ARM instructions are executed
Time 'X'
ARM software tries to read from the externally mapped address
Callback function gets data from external behavior
Simulator sees the access as if it happened at t = 0
waitfor(quantum)
error
External mapped behavior
Simulation time 'X'
SpecC simulation time = quantum
ARM instructions are executed
Host machine execution time

Figure 3.4: Timing error in bus accesses due to large quantum.

The software executing in the ARM CPU tries to access an externally mapped bus register at some execution time "X" during the call to *icmSimulate*. Through the SpecC callback mechanism, these bus accesses are all executed at the beginning of a quantum. The SpecC *waitfor*() is called only after quantum time ARM instructions are executed in the OVP simulator. This makes the access to the externally mapped behavior in the

SpecC simulator happen only at the beginning of quantum boundaries, and not at time "X". Therefore, any differences between quantum – "X" and the real delay becomes a timing error.

A similar issue is presented when interrupts are signaled asynchronously from external hardware or peripheral behaviors into the CPU model (See Figure 3.5). Interrupts will only be recognized by OVP at each call to *icmSimulate*. In this case, contrary to the read/write bus accesses, interrupts will be serviced later than they should. This would add erroneous delays to the simulation time.

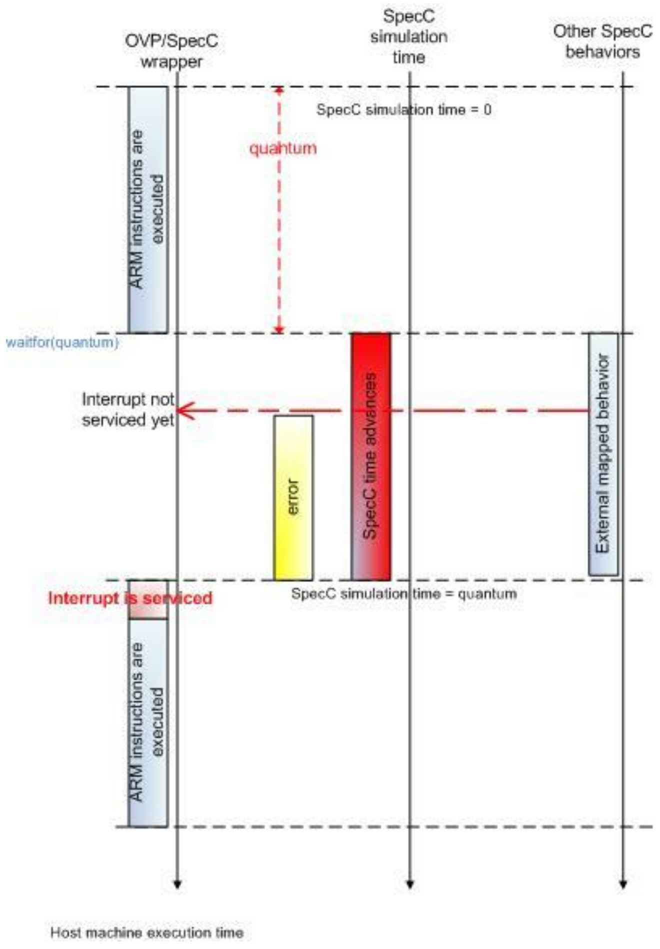

OVP/SpecC wrapper
SpecC simulation time
Other SpecC behaviors
SpecC simulation time = 0
quantum
ARM instructions are executed
waitfor(quantum)
Interrupt not serviced yet
error
SpecC time advances
External mapped behavior
Interrupt is serviced
SpecC simulation time = quantum
ARM instructions are executed
Host machine execution time

Figure 3.5: Timing error in asynchronous interrupt due to large quantum.

It becomes obvious that the quantum value should be carefully selected, and the right value for it could vary from application to application. The resulting errors add up to already existing errors in the ISS introduced as oversimplifications of the approximate timing model. On the other hand, utilization of a too small quantum value might take away any advantages of the fast simulator, since the frequent context switching would bring the simulation speed down, making the platform simulation almost as slow as the cycle accurate simulator (but still not as precise).

As part of this report we wanted to verify if our approach was still capable of producing approximate timing results even when facing these potential issues.

3.2 RTOS, CROSS-COMPILERS AND SOFTWARE SYNTHESIS SUPPORT

During the refinement process in the SCE framework, designers may map several behaviors to a single Processing Element (PE). That means that some sort of firmware should schedule behaviors statically or dynamically to run in the chosen CPU according to the selected priorities. This firmware would mostly come in the form of a RTOS.

At the software synthesis stage, SCE generates the C code that represents the behaviors running in the PE. Through a cross compiler tool chain, the final executable code is linked against the ported RTOS to create the final target binary.

Therefore, in order to be able to run concurrent behaviors in a single PE, SCE needs to provide the following:

- Synthesis into C code of the SpecC behaviors code.
- A ported RTOS to be run in the selected CPU or Platform.
- Cross-compilation into the target architecture and linkage against the RTOS.

In order to minimize unexpected behavior, we decided to test our SpecC/OVP wrapper by running applications on the bare metal without any RTOS. All behaviors in our example application that are mapped to the same CPU run sequentially. However, due to some disparities between the cycle accurate (SWARM) and ISS (OVP) models, we still had to modify some of the no-RTOS support in SCE. Elements like CPU initialization, Interrupt Registration Routines and Interrupt Service Routines were adapted using OVP's example code of their integration with SystemC. Overall, although we limit our current work to bare-metal applications, we established a baseline for future projects. In the next chapter, we present our example application, experiments and results.

CHAPTER 4

Experiments

In this section we describe our example application, SCE refinement steps taken, and simulation results obtained.

For the example application, we have chosen to test the model of a digital camera written in SpecC. Going through all the SCE refinement processes, we generated a working C code and target binary to be tested against our SpecC/OVP wrapper.

4.1 EXAMPLE APPLICATION

In order to properly test the advantages of our approach, we need to choose a proper application which could be executed on the ARM CPU. Specially, in order to prove the benefits of our approach, the choice of the application should be measured to have a high processor workload.

We have chosen to emulate the basic blocks of a digital camera with its corresponding JPEG encoder at the core. The algorithm for DCT based JPEG encoding works as follows [30]:

- Divide the image into 8*8 blocks and do the following for each block:
    - Shift the block and perform a Discrete Cosine Transform (DCT)
    - Quantize the block
    - Perform entropy coding
        - Arrange the block components in a “zigzag” order
        - Huffman encoding
    - Write the encoded information to the output

Ideally, all three major JPEG encoding steps (DCT, Quantize and Entropy encoding) should be performed in parallel. But, in order to run these processes in a CPU with no RTOS, behaviors had to be serialized. Figures 4.1 and 4.2 show both designs, the original and serialized JPEG models. The figure for the serialized model (figure 4.2) does not include the communication channels for readability.

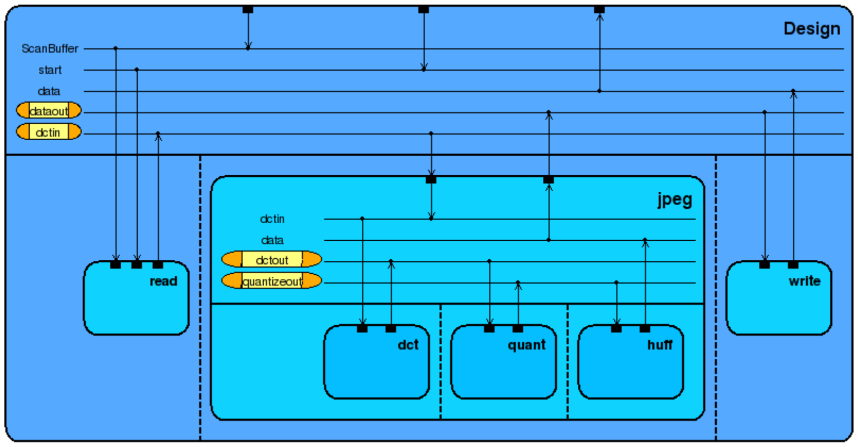


Figure 4.1: Paralleled representation of the Digital Camera example in SpecC notation.

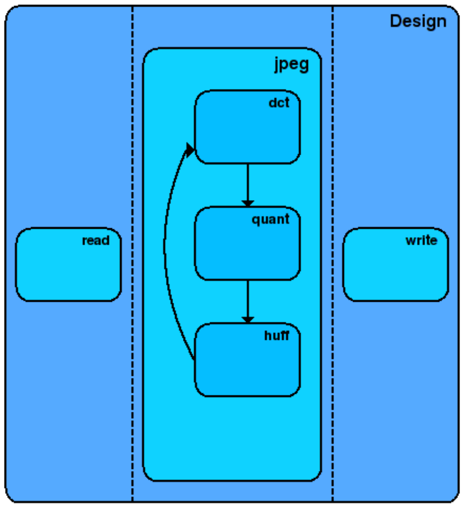


Figure 4.2: Serialized representation of the Digital Camera example in SpecC notation.

In this digital camera example, there are three top level blocks: Read, JPEG and Write. The Read block reads the input data from a ".bmp" file and writes all pixels info into a buffer. Then, the JPEG top behavior grabs the data one block at a time, and sends it to the Write behavior. All combined, the read block emulates the functionality of a CCD. The JPEG behavior represents the encoding unit, and the write block emulates the storage memory.

The JPEG block has 3 main leaf behaviors: DCT, Quantize, and the Huff encoder that conform to the JPEG standard. Notice that although all 3 JPEG sub-blocks are serialized (represented by a FSM as shown in Figure 4.2), the top-level Read, JPEG and Write blocks are all run in parallel.

4.2 REFINEMENT

By using SCE's top-down refinement procedures, we go from the specification of our digital camera example (written in SpecC) to an instruction accurate and bus-functional version of the system. Simultaneously, we test a cycle accurate version of the system that will become our golden reference. At the end, our system should look as shown in Figure 4.3.

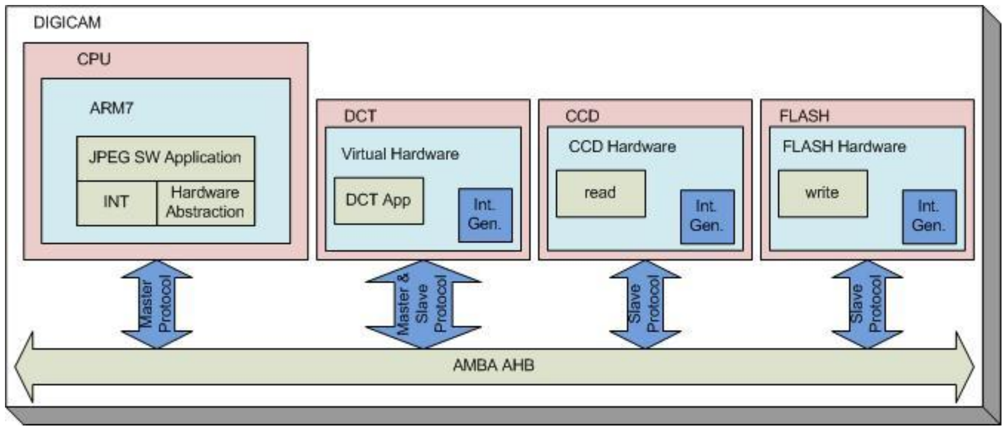


Figure 4.3: Digital camera architecture.

In the first refinement step, we mapped each of our design behaviors to a PE from the SCE database. At this level, computational blocks are still purely behavioral. At the next step, we skipped the scheduling since we didn't need to include a scheduler. Tasks in the CPU should be executed sequentially as defined in Figure 4.2.

Successively, we proceeded with allocating the AMBA bus and connected each PE either through a master or slave interface as indicated in Figure 4.3. For communication synthesis, we mapped each component in the bus address space as follows:

- DCT: 0x10000000 - 0x10000003
- CCD: 0x20000000 - 0x20000003
- FLASH: 0x30000000 – 0x30000003

Interrupt lines were assigned to each peripheral accordingly. We then generated Transaction Level and Pin Accurate models (TLM/PAM) of our system.

Finally, we synthesized the software and replaced the behavioral model of the CPU with our OVP wrapper. The final result is a bus-functional system that co-simulates hardware and software, which provides fast, instruction-accurate execution of the software and communication.

During all stages of the refinement, we checked that the refined models were correct by comparing the JPEG encoded output with a golden file. Also, it should be noted that all the above steps are done interactively through a GUI, which makes the whole process relatively easy.

4.3 RESULTS

We ran each refined model 10 times, averaged the execution time and validated the correct functional execution on all of them. All experiments were run on the UT-Austin compute environment, which uses dual-core 3GHz Xeon servers.

We used three different quantum values when testing the ISS OVP model. We also ran the SWARM cycle accurate model (averaged over 10 times) to compare its execution time against ours. Results are presented in Table 2.

Model	**Simulation reported encoding time (us)**	**Execution time (seconds)**
Spec	-	0
Architecture	-	0
Schedule	-	0
Network	-	0.01
TLM	4,637	1.392
PAM	4,629	14.476
PAM(ISS) + C [quantum = 1ms]	363,037	14.726
PAM(ISS) + C [quantum = 0.1ms]	150,715	15.599
PAM(ISS) + C [quantum = 0.08ms]	139,772	15.778
PAM(SWARM) + C	90,685	183.930

Table 2: Model simulation and execution times.

Only starting from the TLM and PAM models, the SpecC simulator outputs some timing statistics. This is due to the fact that higher levels of abstraction are purely behavioral (computation and communication take zero simulation time). Likewise, computational delays are only taken into account after the ISS or cycle accurate models of the CPU replace the behavioral model of the ARM. By replacing the purely behavioral CPU model with our wrapper into the PAM, we obtain at least an approximate timing if quantum value is selected accordingly.

The execution time up to the TLM model is negligible. Only after inserting the instruction accurate model of the CPU, the execution time goes up considerably. We compared our simulation speedup against the SWARM-based cycle-accurate simulation. It can be seen that our ISS wrapper provides 10+ times faster functional verification than using the cycle-accurate model.

Quantum	**x faster than cycle-accurate (SWARM)**	**x slower than PAM**	**Error**
1 ms	12.49	1.017	300%
0.1 ms	11.79	1.077	66%
0.08 ms	11.66	1.089	54%

Table 3: OVP wrapper against cycle accurate and PAM models.

CHAPTER 5

Conclusion

In this work, we extended our System-Level Design framework to support fast and functional binary-translating ISSs. It achieves 10x faster simulations than using cycle-accurate models for verifying synthesized code.

Our improved System-Level Design tools give embedded system designers flexibility on each level of abstraction to perform hardware exploration and fast prototyping.

An extra positive outcome of our work is the fact that the ISS wrapper can be used for estimating target CPU latencies at the cost of a small increase in simulation time when compared against Pin Accurate and Transaction Level (host compiled) Models.

Additionally, we have laid down the basis for future fast MPSoC simulation and possible automatic generation of an RTOS that could also be functionally verified in our System-Level Design framework.

We plan to expand our work in the future in the aforementioned directions and to add support for other CPU models. Finally, we also intend to investigate possible ways of minimizing the introduced timing error due to improper selection of the simulation quantum.

Finally, we will consider building a similar wrapper for QEMU in order to compare those results to our current ones.

Appendix

QEMU AND OVP COMPARISON OF KEY FEATURES

	QEMU	OVP
Target Processors	Intel x86; PowerPC; ARM7, ARM9E, ARM10E, ARM Cortex, ti925t, pxa270; SPARC and Microblaze.	ARC; ARMv4 and ARMv5; ARM7 thru ARM10; MIPS32 (4K, 24K, 34K, 1004K and 74K); NECel v850; OR1K; and various Tensilica Diamond Cores.
Host Processors	x86, x86_64, and PowerPC.	X86.
Host OS	Windows, Linux, and Mac OS X.	Windows and Linux.
Target Platform OS	Linux 2.6 SPARC; Debian 'Etch' for MIPS; Linux 2.6 for ARM based supported boards; and uClinux for ColdFire based supported boards.	ucLinux for OR1K and AtmelAT 1SAM7; Linux 2.6.17 for ARM IntegratorCP; and Linux for MIPS Malta.
Full Platform Examples	ARM IntegratorCP, VersatilePB, VersatileAB, RealView, Akita PDA, Spitz PDA, Borzoi PDA, Terrier PDA, Palm Tungsten, Stellaris LM3S811EVB, LM3S6965EVB, Gumstix, MIPS Malta, R4K platform, Acer Pica	ARM IntegratorCP, AtmelAT91SAM7, and MIPS Malta.

	61, Heathrow based PowerMAC, Mac99, PowerPC PREP, Sun SPARCstation 5, SPARCstation 10, and many others.	

References

[1] Wolf, W., *High-Performance Embedded Computing.* Morgan Kaufmann Publishers, 2007

[2] Gajski, D.D., Abdi, S., Gerstlauer, A. and Schirner, G., *Embedded System Design: Modeling, Synthesis, and Verification.* Springer US, 2009

[3] Eles, P., Kuchcinski, K. and Peng., P., *System Synthesis with VHDL*. Kluwer Academic Publishers, December 1997

[4] Lilja, D. and Sapatnekar, S., *Designing Digital Computer Systems with Verilog*. Cambridge University Press, December 2004.

[5] Grötker, T., et al., *System Design with SystemC*. Kluwer Academic Publishers, 2002

[6] SpecC Technology Open Consortium, http://www.specc.gr.jp/eng/index.htm

[7] Gajski, D.,*"SpecC : specification language and methodology*. Kluwer Academic Publishers, 2000

[8] Cai, L., Verma S. and Gajski, D., "Comparison of SpecC and SystemC Languages for System Design". University of California, Irvine. Technical Report, 2003

[9] Domer, R. et al., "System-on-Chip Environment: A SpecC-Based Framework for Heterogeneous MPSoC Design". Research Article. University of California, Irvine, 2008

[10] Schirner, G., Sachdeva, G., Gerstlauer, A. and Rainer D., "Modeling, Simulation and Synthesis in an Embedded Software Design Flow for an ARM Processor". University of California, Irvine. Technical Report, 2006

[11] SWARM - Software ARM, http://www.cl.cam.ac.uk/~mwd24/phd/swarm.html

[12] Bellard, F. "QEMU, a Fast and Portable Dynamic Translator", Proceedings of the annual conference on USENIX Annual Technical Conference, pp. 41 - 41, 2005

[13] Wang, M. et al., "System-Level Development and Verification Framework for High-Performance System Accelerator", National Cheng Kung University, Tainan, Taiwan, 2009. http://caslab.ee.ncku.edu.tw/research/publications/CASLab_2009_CNF_01.pdf

[14] Monton, M. et al., "Mixed SW/SystemC SoC Emulation Framework". IEEE International Symposium on Industrial Electronics, pp. 2338-2341, 2007

[15] QEMU-SystemC. http://www.greensocs.com/en/Projects/QEMUSystemC

[16] Open Virtual Platforms (OVP), http://www.ovpworld.org/index.php

[17] The Open SystemC Initiative (OSCI), http://www.systemc.org/downloads/standards/

[18] SPEC. Standard performance evaluation corporation. http://www.spec.org

[19] Perelman, E., Hamerly, G., Calder, B., "Picking statistically valid and early simulation points". International Conference on Parallel Architectures and Compilation Techniques, September 2003

[20] Weaver, M., McKee, S., "Are Cycle Accurate Simulations a Waste of Time?", Seventh Annual Workshop on Duplicating, Deconstructing, and Debunking, Beijing, China, June 2008

[21] Abdi, S. et al., "System-on-Chip Environment", University of California, Irvine, Technical Report, 2003

[22] Chiou, D. et al., "The FAST Methodology for High-Speed SoC/Computer Simulation", International Conference on Computer Aided Design, San Jose, California, 2007
[23] Virtual Box, http://www.virtualbox.org/
[24] VMware, http://www.vmware.com/
[25] Kernel Based Virtual Machine, http://www.linux-kvm.org/page/Main_Page
[26] VDC Embedded Systems Practice, "Imperas Forms the OVP (Open Virtual Platform) Initiative", March, 2008
[27] Bailey, B., "System Level Virtual Prototyping becomes a reality with OVP donation from Imperas", White Paper, June 2008
[28] "OVPsim and Imperas CpuManager User Guide", Imperas Ltd., Version 2.0.11
[29] Labrosse, J., *MicroC/OS-II: The Real-Time Kernel.* CMP Books, 2002
[30] CCITT Recommendation T.81 on Information Technology – "Digital Compression and Coding of Continuous Tone Still Images - Requirements and Guidelines", ISO/IEC 10918-1, September 1992, http://www.w3.org/Graphics/JPEG/itu-t81.pdf
[31] Daedalus, http://daedalus.liacs.nl/Site/Daedalus%20home.html
[32] Metropolis, http://embedded.eecs.berkeley.edu/metropolis/
[33] Keinert J., et al., "SystemCoDesigner—an automatic ESL synthesis approach by design space exploration and behavioral synthesis for streaming applications", Transactions on Design Automation of Electronic Systems, January 2009
[34] Synphony HLS, http://www.synopsys.com/Tools/SLD/HLS/Pages/default.aspx
[35] Catapult-C synthesis, http://www.mentor.com/products/esl/high_level_synthesis/
[36] Lavagno, L., Sangiovanni-Vincentelli, A., Sentovich, E., "Models of computation for embedded system design", System Level Synthesis, pp 45-102, Kluwer Academic Publishers, 1999
[37] Smith, J., Nair, R., "The Architecture of Virtual Machines", IEEE Computer, pp. 32-38, May 2005.
[38] Chung, M. K., et al., "Improvement of Compiled Instruction Set Simulator by Increasing Flexibility and Reducing Compile Time", 15th IEEE International Workshop on Rapid System Prototyping, June 2004
[39] Reshadi, M., et al. "Instruction Set Compiled Simulation: A Technique for Fast and Flexible Instruction Set Simulation", Proceedings of Design Automation Conference, 2003