Universitat Autònoma de Barcelona

Escola d'Enginyeria
Departament de Microelectrònica i Sistemes Electrònics

Checkpointing for Virtual Platforms and SystemC-TLM-2.0

Memòria del treball de tesis del programa
de Doctorat en Informàtica,
opció microelectrònica,
presentada per **Màrius Montón i Macián**
i dirigida per Dr. Mark Burton
i Dr. Jordi Carrabina i Bordoll

Bellaterra, 10 de Novembre del 2010.

To my parents and sister

To Míriam

Acknowledgements

I would like to thank all the people who have helped and inspired me during my doctoral study these years.

I especially want to thank Dr. Carrabina for the years working with him in CEPHIS at Universitat Autònoma de Barcelona.

Dr. Burton and Dr. Jakob Engblom deserve special thanks for their guidance during work with them at GreenSocs. This thesis wouldn't be possible without them: thank you. Thanks also to the other GreenSocs staff, I've learned a lot from all of them.

Very special thanks to Josep Cañadell. He was the student who started the technical work presented here as his Final Degree Project.

Specially thank to CEPHIS guys, started by Borja, Willy & Toni and continuing with Marc, Aitor, Roger, Chak, Carlos, Edu, Héctor, and all the rest of the group. All of them deserves the same opportunities to get their PhD as I had.

Also thank to my teaching colleagues Xavi Fitó, Lena, Quim, Elena Valderrama, Lluís Ribas, I learned a lot from them, I hope they get something from me some time.

There are lot of people from UAB that I must be grateful to. Elena Garcia, Obi, Maria, Albert, Anna, Pilar, Toni Nuñez, Otto, Guille, thank you for the coffees, lunches, dinners, beers, inspiration, comments...

I wish to thank everybody with whom I have shared experiences in life. From the people who first persuaded and got me interested in the study of informatics, especially those who also played a significant role in my life, to those who with the gift of their company made my days more enjoyable and worth living: Bosko, Roman, Marta, Manolo, Ben, Xavifú, Gemma, Héctor, Àlex, Chema, Sandra and other GTR people; inthenight nerds; Albert Guinart, Nadia, Albert Prats, Mónica... you probably don't understand any of this thesis, but it is done in great part thank to you.

I cannot finish without saying how grateful I am with my family: grandparents, uncles, aunts and cousins and 'acoplaos' all have given me a loving environment in which to develop. I wish to thank my parents, Dolors and Faustí and my sister Núria. They have always supported and encouraged me to do my best in all matters of life. To them I dedicate this thesis.

Of course, Míriam deserves a special line to herself. Without her patience, support and love, this work wouldn't have been completed. Thank you to my new family, I feel at home with them.

Lastly, I offer my regards to all of those who supported me in any respect during the completion of the project and whom I may have forgotten in these acknowledgements.

Abstract

One advantage of using a virtual platform or virtual prototype over real hardware for embedded software development and testing is the ability of some simulators to take checkpoints of their state.

If the entire system model is detailed enough, it might take several minutes (or even hours) to simulate booting the O.S. If a snapshot of the simulation is saved just after it has finished booting, each time it is necessary to run the embedded software, designers can simply restore the snapshot and go. Restarting a checkpoint typically takes a few seconds. This can translate into a major productivity gain, especially when working with embedded system with complex SW stacks and O.S. like modern embedded devices.

In this dissertation we present in firstly our work on adding a description level language as SystemC to two Virtual Platforms. This work was done for a commercial Virtual Platform, and later translated to a open-sourced Platform.

This thesis also presents a set of modifications to SystemC language to support checkpointing. These modifications will make it possible to take the state of a SystemC running simulation and save it to disk. Later, the same simulation can be restored to the same point it was before, without any change to the simulated modules. These changes would help SystemC to be suitable for use by Virtual Platforms as a description language.

Finally, we discuss and propose some other alternatives to enhance SystemC in the integration to Virtual Platform and to other EDA tools.

Contents

ACKNOWLEDGEMENTS....VII

ABSTRACT....IX

LIST OF FIGURES....XVII

INDEX OF TABLES....XIX

1 INTRODUCTION....23

1.1 Objectives....25

1.2 Contributions....25

1.3 Structure of the dissertation....26

2 STATE OF THE ART....29

2.1 SystemC....29

2.2 TLM-2....30

2.2.1 Transport Interfaces....33

2.2.2 Coding Styles....35

2.2.3 Temporal decoupling....36

2.2.4 Direct Memory Interface....37

2.3 Virtual Platforms....38

2.3.1 Instruction Set Simulators....39

2.3.2 Current tools....40

2.4 SystemC and Virtual Platforms....48

2.5 Checkpointing....49

2.6 Checkpointing in Virtual Platforms....51

3 VIRTUAL PLATFORMS AND SYSTEMC....55

3.1 QEMU-SC....55

3.1.1 Motivation....55

3.1.2 Architecture....56

3.1.3 Implementation....59

3.1.4 Results....63

3.2 Virtutech Simics Bridge..........69
3.2.1 Architecture..........69
3.2.2 Implementation..........71
3.2.3 Results..........73
3.3 QBox..........75
3.3.1 Architecture..........76
3.3.2 Implementation..........77
3.3.3 Results..........79
4 CHECKPOINTING FOR SYSTEMC & TLM-2..........83
4.1 Motivation..........83
4.2 Related Work..........83
4.3 Design and Implementation..........85
4.3.1 Model requirements..........86
4.3.2 SystemC kernel..........88
4.3.3 FSM..........97
4.4 Results..........99
4.4.1 SC_THREAD to SC_METHOD overhead..........101
4.4.2 SystemC checkpoint support overhead..........101
4.4.3 Checkpoint size..........103
4.4.4 Validating Checkpointing..........105
4.4.5 Validating Model Updates..........106
4.4.6 Completeness of checkpointable SystemC subset..........107
4.4.7 Simulation speed of the checkpointable OSCI kernel..........109
4.5 Limitations and Discussion..........109
5 DISCUSSION AND CONCLUSIONS..........113
5.1 Discussion..........113
5.2 Conclusions..........114
5.3 Impact of the QEMU & SystemC..........116
CURRICULUM VITAE..........CXXI
PUBLICATIONS..........CXXIII
REFERENCES..........CXXIX
ALPHABETICAL INDEX..........CXXXVII

List of Figures

Figure 2.1: SystemC Structure..30
Figure 2.2: RTL (a) vs TLM (b) modeling....................................31
Figure 2.3: TLM-2 Coding Styles..35
Figure 2.4: Simulation of 3 steps for 3 modules 36
Figure 2.5: Direct Memory Interface...37
Figure 2.6: Block diagram of a generic Virtual Platform..................39
Figure 2.7: Virtual Platform: Emulated Android device...................42
Figure 3.1: Block Diagram (SystemC blocks grayed)........................57
Figure 3.2: QEMU – SystemC Dataflow..58
Figure 3.3: Synchronization pseudo-code....................................61
Figure 3.4: IRQ generation code..65
Figure 3.5: Final test bench module..66
Figure 3.6: Simics and SystemC Bridge 70
Figure 3.7: Communication schema..72
Figure 3.8: Simics bridge block diagram......................................73
Figure 3.9: Wrapped QEMU block diagram..................................75
Figure 3.10: Complex test bench block diagram.............................79
Figure 3.11: QBox in Innovator..81
Figure 4.1: Example use of gs_param<>......................................86
Figure 4.2: Pseudocode for do_checkpoint() function.....................96
Figure 4.3: FSM example..97
Figure 4.4: Code example for a SC_THREAD.................................98
Figure 4.5: Code example for a FSM using SC_METHOD.................98
Figure 4.6: Checkpoint information...103
Figure 4.7: Code for test device...105
Figure 4.8: Typical VP device...107
Figure 5.1: QEMU Ecosystem..117

Index of Tables

Table 2.1: Performance test results for emulator and real devices.....44
Table 3.1: Execution time for different MPEG-2 decoder...................68
Table 3.2: Results for UART experiments on Simics..........................74
Table 4.1: Time results for FSM description....................................100
Table 4.2: Performance test result for gs param..............................102

1 Introduction

Nowadays, with the complexity of devices that the market demands, in great part thanks to the increase in density of the current hardware, the necessity for a new generation of simulation tools is apparent.

These new tools may be capable to simulate anything from simple systems like a microprocessor with a small set of peripherals running an embedded single application to very complex and large systems like a heterogeneous multi-core rack of processors connected through a standard network infrastructure running a parallel program using some HPC[1] libraries and O.S.

It seems obvious that these kinds of new tools cannot use RTL models, because of its natural slowness. 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. However, 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.

1 High Performance Computer.

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

On the other hand, SystemC has become a standard language applied to system-level modeling, architectural exploration, performance modeling, software development, functional verification and high-level synthesis. Recent years have been associated with Transaction Level Modeling (TLM).

But at the time of writting, no Virtual Platform supported SystemC as a language to describe models of the platform.

This lack of support was needed to be fixed using the TLM strategy to achieve great performance in simulations.

1.1 Objectives

The main objective of this work is to add SystemC capabilities to some existing Virtual Platforms: we work with an open-sourced virtual platform named QEMU and a commercial tool from Virtutech named Simics.

To join two different simulators (Virtual Platform with SystemC kernel) we used some synchronization strategies following TLM-2 rules to get the maximum performance, portability and standardization.

Related to the habitual use of Virtual Platforms, we introduce a new feature to SystemC simulator called checkpointing as a key-enabler for this language to be useful for Virtual Platform tools users.

1.2 Contributions

Major contributions of this work are summarized in this section. Deeper and more detailed analysis is done in chapter **5 Discussion and Conclusions.**

- **Adding SystemC capabilities to Virtual Platform**: we worked with two different Virtual Platforms to enable them to use SystemC and TLM-2 modeled devices. This work included the major task of synchronizing two simulators, managing communications between the two realms, and the development of a complex test system to check all functionality.
- **Adding checkpointing to SystemC** to make this characteristic usable in the Virtual Platforms with SystemC. This characteristic will allow Virtual Platforms to stop, save, and restore simulations in order to improve productivity when working with large systems.
- Giving a **set of rules to write SystemC models** capable of being used in a Virtual Platform supporting checkpoint. Our solution for SystemC checkpointing wouldn't be complete, and some limitations and guidelines will be necessary to write models for use with it.

1.3 Structure of the dissertation

The following chapters illustrate the issues and problems presented in this introduction:

- **Chapter 2 State of the Art** analyses existing commercial and free open sourced Virtual Platforms, and their main characteristics. We introduce in deep detail the problem of checkpointing for Virtual Platforms, and the use of SystemC on Virtual Platforms.
- **Chapter 3 Virtual Platforms and SystemC** presents the methodology used to add SystemC support to different Virtual Platforms, one open-sourced and a commercial one.
- **Chapter 4 Checkpointing for SystemC & TLM-2** introduces the work on SystemC simulator kernel to make it capable of checkpointing when used to model different parts of a Virtual Platform.
- **Chapter 5 Discussion and Conclusions** summarizes the results and exposes global conclusions. Also, limitations and possible future research are discussed.

2 State of the Art

In this chapter we present the current state of the Virtual Platforms field, major companies working in this topic, their tools and their features, along with a brief introduction to SystemC and Transaction Level Modeling (TLM). To end, the chapter will deal with a typical feature for Virtual Platforms named Checkpointing, and the listing of the different Virtual Platform tools that support SystemC.

2.1 SystemC

SystemC is a simulation language standardized as IEEE 1666 [1]. It is maintained and developed by the OSCI (Open SystemC Initiative)[2]. The version of the base language as of the time of writing is 2.2.0.

SystemC is built on C++, and provides a set of classes and templates to facilitate the simulation of digital systems. SystemC provides support for coding hardware concepts like parallel units communicating across channels, data types that can handle four-valued logic and arbitrary length integers, and a scheduler that drives the parallel simulation summarized in *Figure 2.1*.

SystemC grew out of the need from chip designers for a level of simulation and modeling above the RTL implementation expressed in VHDL or Verilog. SystemC aims to provide a single language for the entire design flow from design to implementation.

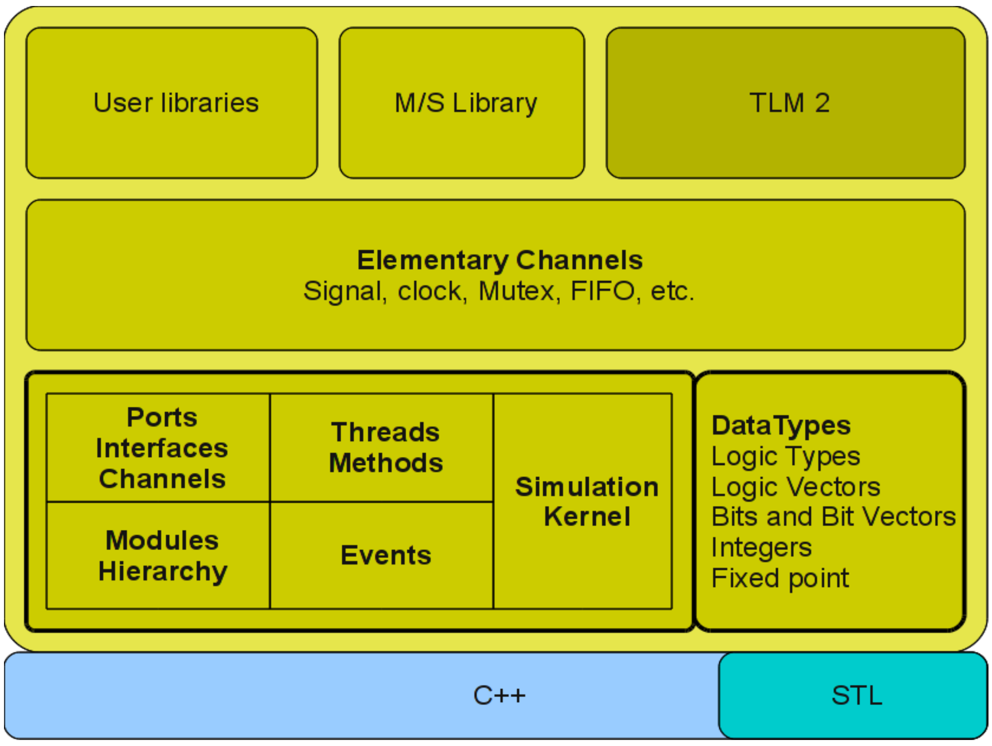


Figure 2.1: SystemC Structure

In addition to the base language, OSCI also has working groups working on extensions and specifications for SystemC. There is no working group specifically for virtual prototypes, but the working group on Transaction-Level Modeling (TLM) has simulation for the benefit of software development as one of its primary use cases [3].

2.2 TLM-2

In Transaction Level Modeling, hardware models are designed at a higher level than the typical detailed models used to optimize and validate hardware designs. This is achieved by changing the style of communication between hardware units from individual bits being clocked out on communication channels one cycle at a time to entire “transactions”. The overall idea is to make sure to communicate only the information needed to actually simulate what is happening, but not necessarily exactly how it is being performed in the hardware implementation.

It is also important to perform this communication in an efficient manner, typically by trying to achieve each transaction in a single function call in the simulator. As illustrated in *Figure 2.2*, the goal is to do this across an interconnect in such a manner that the interconnect does not need to perform any kind of work to process the transaction.

There is no precise definition of what constitutes a transaction, but typical examples from the world of virtual prototyping are:

- Memory reads and writes between a processor and device.
- Raising an interrupt.
- Resetting a system, considered as an atomic action.

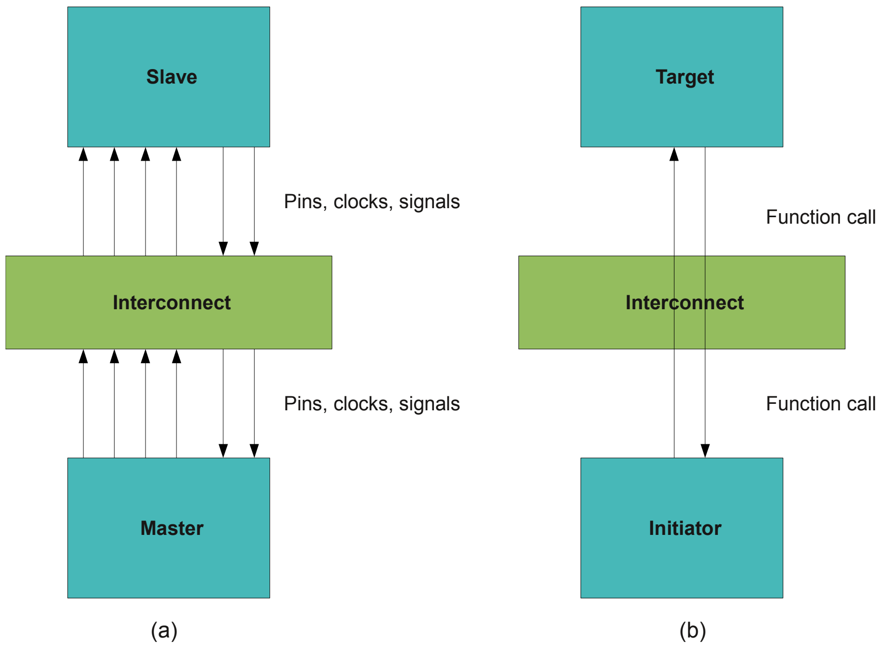


Figure 2.2: RTL (a) vs TLM (b) modeling

- Sending a character from a serial port towards a display.
- Putting a network packet on a network.
- Delivering a network packet to its target device.
- Capturing a single reading from an analog-to-digital converter.

Examples of hardware activity typically not considered to be transactions:

- Sending a clock cycle into a device, for the purpose of advancing its internal state.
- Internal state changes that do not result in any output visible to other devices. For example, incrementing time in a clock.
- Clocking out the complete sequence of bus signals needed to actually send data over a real-life data bus.
- Bus arbitration before actually getting access to a bus.

Often, there can be several different levels of abstraction for the same event in terms of transactions. For example, a memory read could be broken up into a request transaction and a response transaction for a coarse model of a pipelined bus.

From the perspective of building a virtual prototype, transactions have to capture all the information necessary to expose correct function towards the software and nothing else.

Also note that in terms of Virtual Platforms, a transaction can be seen as an abstraction of any of the memory mapped buses found on habitual systems, like PCI [4], AMBA [5], CoreConnect [6], etc., because all those buses mainly use a (address|data|command) 3-tuple for communication.

This feature will help us to write devices that can be plugged into any of these platforms without notables changes in the code, abstracting the fabric bus used and replacing it with a TLM-2 model of the bus.

2.2.1 Transport Interfaces

TLM-2 defines a set of coding styles to help developers to improve interoperability of the models. These coding styles are based on the use of two different interfaces: **blocking** and **non-blocking** function calls.

2.2.1.1 Blocking

In the blocking interface, all communication between two modules is done using a single function call (*b_transport()*). This function is called at Initiator side, and implemented in the Target module.

The target device can call *wait* to spent simulation time in the body of *b_transport* function. In this way, a transaction can advance simulation time if the Target calls *wait()* when responding to the transaction.

This mechanism has implicitly only two timing points, one at function call and the other at function return. The transaction ends with the return of the function *b_transport*.

2.2.1.2 Non-blocking

In this interface, the communication between the two modules is done in an asynchronous way using the *nb_transport()*[2] method. That is, the function may return immediately, without any option to call wait() or yield control to the simulator in any other form.

Using non-blocking interface splits the communication into its phases, corresponding each phase to the call to the corresponding *nb_transport()* method.

2There is a nb_transport for each agent of the communication: *nb_transport_fw()* is called by the Initiator and *nb_transport_bw()* is used by the Target.

2.2.2 Coding Styles

OSCI TLM WorkGroup provides different definitions for various levels of time accuracy to use with Transaction Level Modeling.

2.2.2.1 Untimed

This style is defined in the following manner by the Working Group:

"A modeling style in which there is no explicit mention of time or cycles, but which includes concurrency and sequencing of operations. In the absence of any explicit notion of time as such, the sequencing of operations across multiple concurrent threads must be accomplished using synchronization primitives such as events, mutexes and blocking FIFOs. [...]" [7]

This means that time in this simulation model can be completed in zero virtual time. In principle, this coding style is not suitable for Virtual Platforms, since software usually expects that certain types of hardware take some time. The trivial example is timers, that they are expected to end after some time, not immediately. A simple example is depicted in *Figure 2.3 (a)*.

2.2.2.2 Loosely Timed

Loosely timed is defined in the following way:

"A modeling style that represents minimal timing information sufficient only to support features necessary to boot an operating system and to manage multiple threads in the absence of explicit synchronization between those threads. A loosely timed model may include timer models and a notional arbitration interval or execution slot length. [...]." [7]

This makes use of the blocking interface, allowing only two points associated with each transaction: the call to the blocking call and the return from it. If using the base protocol, the first timing point marks the beginning of the request, and the second marks the beginning of the response. These two points can occur at the same simulation time or at different times (because Target has called *wait()*). An example is shown in *Figure 2.3* (*b*).

2.2.2.3 Approximately-Timed

This coding style is defined in this way:

"A modeling style for which there exists a one-to-one mapping between the externally observable states of the model and the states of some corresponding detailed reference model such that the mapping preserves the sequence of state transitions but not their precise timing. The degree of timing accuracy is undefined." [7]

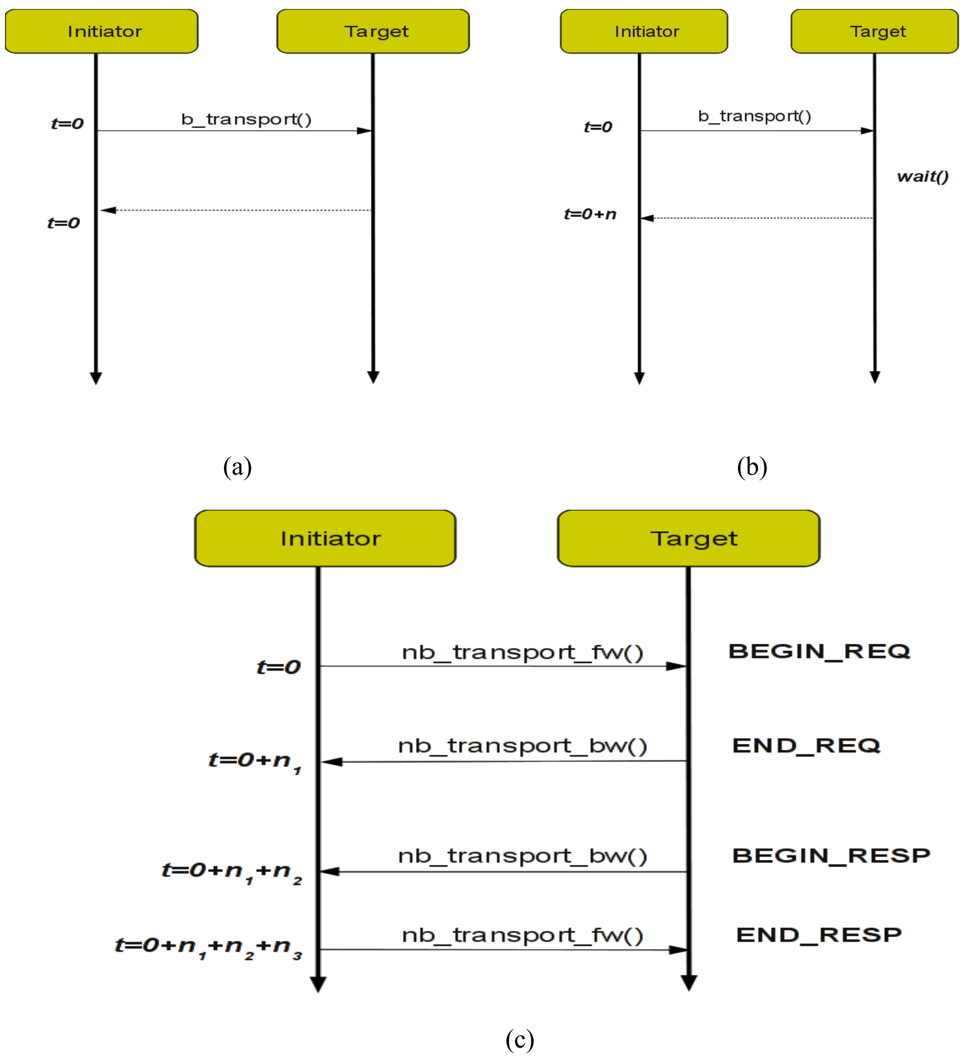


Figure 2.3: TLM-2 Coding Styles: (a) Untimed (b) Loosely-timed (c) Approximately-Timed

This makes use of the non-blocking interface, allowing a non-specified number of points per transaction. The most important characteristic of this style is that these points can correspond to the same events as in an RTL or cycle-timed simulation. Phases and mechanisms are depicted in *Figure 2.3 (c)*.

With this style, designers can very accurately simulate behaviour of buses or any communication mechanism and observe successes like bus contention, bus arbitration, etc.

2.2.3 Temporal decoupling

Also, to boost simulation performance, TLM-2 offers Temporal Decoupling to re-use locality of execution in the simulator. The concept behind it is that most of the time, simulation parts will not interact very often, and this can be exploited to allow a piece of the simulation to run ahead of the current simulation time for a short amount of time.

The consequence of this technique is that different parts of a system can have minor differences in their local simulation time from the current simulation time. The maximum time that one part of the simulation is allowed to run ahead of the current simulation time is called quantum.

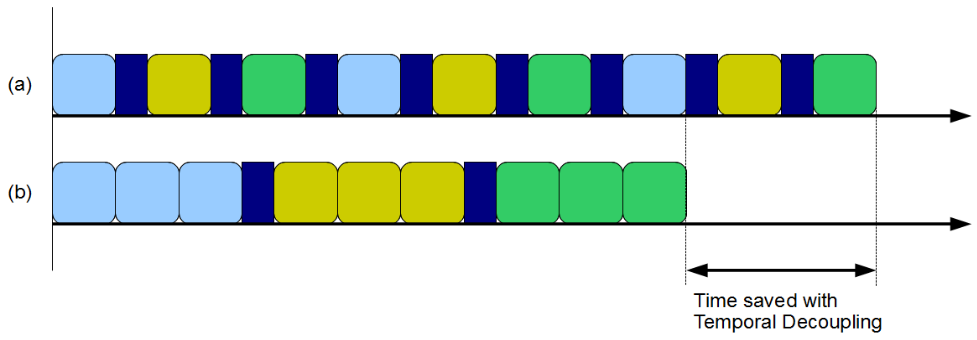


Figure 2.4: Simulation of 3 steps for 3 modules in (a) Synchronized simulation (b) Temporally decoupled simulation

In the field of Virtual Platforms using TLM, hard synchronization is not necessary. Different modules of a system don't interact very often, and this characteristic can be used to improve simulation speed: each couple of Initiator and Target that are communicating between them without interacting with any other module can be simulated for a while ahead of global simulation time as shown in *Figure 2.4*.

2.2.4 Direct Memory Interface

The TLM-2 Direct Memory Interface (DMI) is a specialized interface that provides direct access to a memory area within a target. This interface is intended for memory devices, which will be accessed very often in a simulation.

This interface can accelerate memory accesses because it allows direct access to the memory using a pointer to the region, rather than having to pass through standard socket transactions as depicted in *Figure 2.5*.

The Initiator asks the Target for a DMI pointer of the selected region, and Target responds with the correct pointer. At any time, the Target can invalidate the current DMI pointer using the backward path to notify it to the Initiator, which cannot longer use the pointer to access the memory region in the Target.

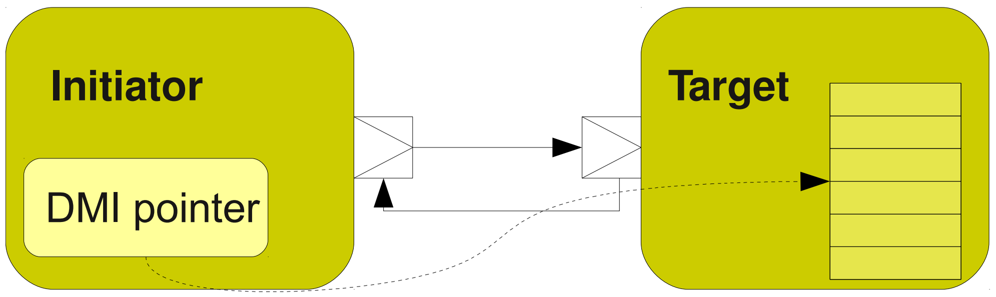


Figure 2.5: Direct Memory Interface

2.3 <u>Virtual Platforms</u>

A virtual system platform is a simulation of a real (or in development) system platform. These simulations usually run standard Operating Systems without needing to change them, and normal applications upon the operating system. In the ESL field, this kind of virtual platform allows early development of the system software (operating system, device drivers, firmware, etc.) without the need for the real Hardware.

The virtual prototype should run the same software as the eventual target system, with no modifications or special cases. This means that the virtual prototype has to model the properties and behaviour of the system in a way that corresponds to how the hardware works at the level where the software talks to the hardware. In practice, this means that a virtual prototype contains four elements (as shown in *Figure 2.6*):

- Processor models for the target processors used, usually an Instruction Set Simulator (ISS).
- Device models for the peripheral units in the target system.
- Memory models for RAM, FLASH, and other memories in the target system.
- Network models for the networks connecting multiple target machines, or connecting the target machines to the outside world.

For a Virtual Platform tool, different choices of these elements create a different emulated platform, sometimes sharing some elements between very different platforms: the same module emulating some specific UART device can be used to emulate an x86 platform or an ARM based board like Integrator/CP [8].

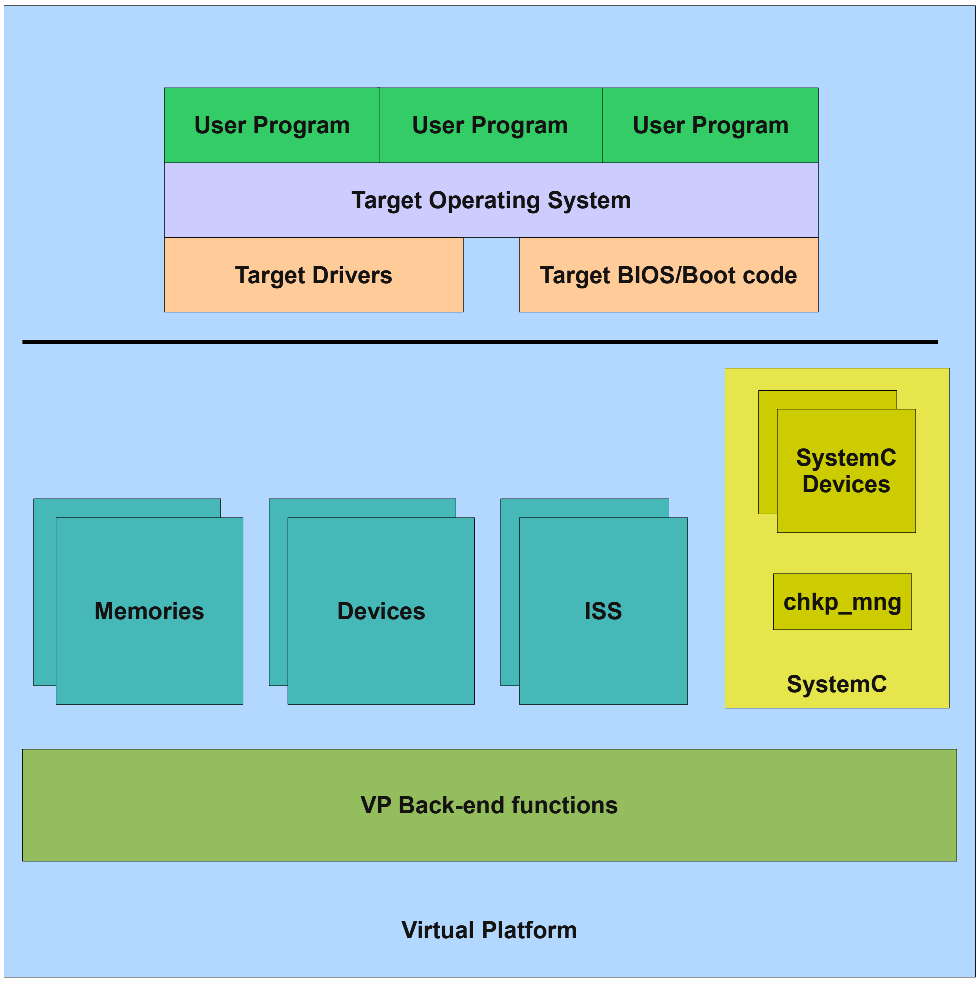


Figure 2.6: Block diagram of a generic Virtual Platform with SystemC

2.3.1 Instruction Set Simulators

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

- Interpretive
- Static compiled
- Dynamic translation

Interpretive ISSs 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 re-translation, 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 [9; 10] and inspecting all of them is beyond the scope of this thesis.

ISSs may be encapsulated into virtual platform emulators, such as QEMU [11], Simics [12] and OVP [13]. Virtual Platform emulators not only consider code translation, but also the modeling of peripherals and other ancillary features of a microprocessor, such as MMUs (Memory Management Unit) , timers, etc. As such, Virtual Platform emulators allow the execution of unmodified guest Operating Systems on them.

2.3.2 Current tools

Several products can be included in the definition of Virtual Platform, most notable are: Virtutech's Simics [12], Imperas' Open Virtual Platform [13], CoWare Virtual Platform [14], Cadence Incisive Simulator [15], Mentor's EDGE SimTest [16], Synopsys' Innovator [17], QEMU [11], Bochs [18], VMware [19], Microsoft Virtual PC [20].

Some of these (e.g. Simics, OVP, EDGE Sim Test, Innovator, Incisive and CoWare Virtual Platform) are development tools designed to help engineers to develop entire systems, giving them lots of traces and debug capabilities for the state of the system or selected devices.

The others are intended to support running a standard O.S. and applications on it (probably a different guest O.S. than host O.S.), giving a very good performance for end-user, but they are not designed to give any information and are not normally used for platform development. Some of the emulators listed (QEMU, Bochs) could be used for both, but to be helpful for platform designers, some extra features are required.

In this thesis we worked with two different Virtual Platforms: QEMU and Simics. We briefly explain these two Virtual Platforms.

2.3.2.1 QEMU

QEMU is a generic machine emulator based on dynamic translation. Currently QEMU emulates the following processors: Intel x86 and x86_64, ARM, SPARC, MIPS, Coldfire (many other platforms are in the development stage). The emulated systems can run unmodified Operating Systems (guest O.S.) like GNU/Linux or MS-Windows XP on different host O.S.

QEMU consists of various modules:

- CPU emulator (ARM, x86, PowerPC, ...)
- Emulated devices (VGA, Ethernet, HD, ...)
- Machine descriptions (PC, PowerMac, VersatilePB, ...)
- Debugger & User Interface.

The communication between the CPU emulator and the emulated devices is done via registered callback functions for each memory region of the system bus. For instance, in ARM emulated systems, every device registers its functions for read and write operations in a memory range of the memory-mapped bus. Then, when the CPU emulator does an access to a memory address, the proper callback function (to the device registered with this address range) is called and the functionality of the device is emulated.

We focus on finding a way to attach devices written in SystemC as emulated devices to QEMU and maintaining as many of the other parts of the virtual platform as required untouched in the emulator (QEMU). So we need a link between the two worlds: Simplistically, a device that registers itself on the address range and when this device is accessed, performs the access to the SystemC device and return back to the CPU emulator the data read or the result of the operation. However the mechanism will also need to arrange to synchronize the different notions of 'time' within both the QEMU emulation world and the SystemC world.

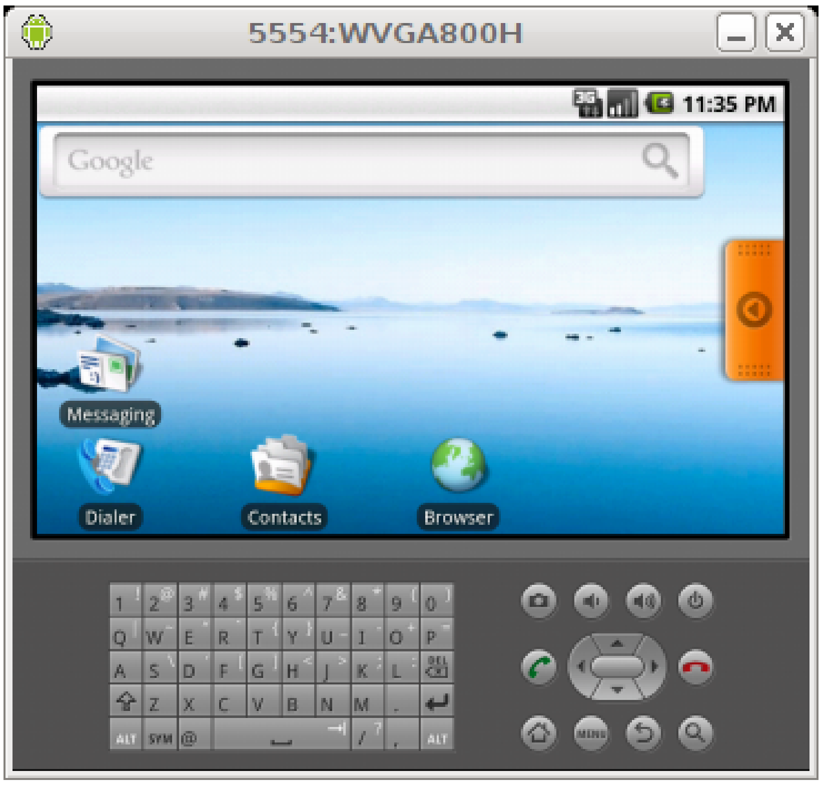


Figure 2.7: Virtual Platform: Emulated Android device (source [22])

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) [21].

Architecture

To allow easy porting of the ISS to new host or target architectures, QEMU first decompiles the simulation binary into fewer and 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 (TB) consist of sequences of target code up to the next jump or instructions 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.

Similar to other emulators, the communication between different emulated devices is done via registered callback functions 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.

Performance

Overall QEMU is a fast, freely available simulator with support for many Operating Systems and target architectures. Android emulator is a modified QEMU [22]; this is a good example of a full virtual platform, used for thousands of developers when they write applications for this O.S.

We summarize a test done with this emulator and different real devices. We used a recursive Fibonacci number calculator to do the tests. As show in Table 2.1, the performance penalty is around 3 times (i.e. the device simulation is about 3 times slower than the real device).

Table 2.1: Performance test results for emulator and real devices

Device	Android version	Execution time (ms)	CPU Mhz	Real Device/ Emulator
Emulator	1.5	316	3000	-
HTC Dream	1.5	97	500	3.2
Emulator	2.1	117	3000	-
HTC Legend	2.1	37	600	3.1
HTC Desire	2.1	30	1000	3.9

2.3.2.2 Simics

Simics is a pure software Virtual Platform. It does not require any special hardware, FPGA boards, or "emulators" to function. Simics runs on any PC, at least as long as the PC in question runs Windows or Linux. Designers can ship a Simics setup literally in an e-mail across the globe. In this way, it can replace hardware boards for global development teams in a fairly magical way.

One large part of a running Simics simulation is the simulation of the target hardware. This contains models of processor cores, buses and other interconnects, memories, peripheral devices, and networks.

A core component of Simics is its fast instruction-set simulators (ISS) which are used to execute the ARM, DSP, MIPS, Power Architecture, SPARC, x86/IA, or other processors' binary code. However, just an ISS is not enough to run an operating system. Simics setups therefore also include simulation models for memory-management units (MMU), as well as all the memories and devices found in the memory maps of the processors.

In a common example, a processor doing a memory operation: when a processor issues a memory load or store, the address to access is first translated through the MMU, resulting in a physical address. This physical address is used to construct a memory transaction in Simics.

The memory transaction is sent to a memory map, which determines what part of the target system the transaction is targeting. If the transaction hits memory, the contents of the memory are read or modified.

If the transaction hits a peripheral device, the simulation model for that device is called to determine the effects of the access. The device model will work out the effects of the access. It might start a timer, send an interrupt to a different processor in the system, reconfigure the hardware, reset a board, send a packet onto a network, drive an I/O pin high, or have any other effect on the target system. If the transaction does not hit anything, the simulated process could get an exception back to report the bus error to the software.

Simics works as a software program on the host PC, a simulation happens as a chain of function calls between the objects making up the simulation model. Everything in a Simics simulation is an object, using function calls to communicate both transactions related to the target system (like memory operations) and for internal operations: maintenance, logging, tracing, and other operations. This style of simulator modeling is very similar to we known as TLM or transaction-level modeling.

Unlike any other virtual platform solution, Simics can add new objects to a simulation at any point in time. It is also possible to reconfigure how the objects are connected, and modify their properties at run-time.

This dynamic configuration makes it possible to simulate anything that can happen in the physical hardware system, including operations like adding and removing boards from a rack or changing how network cables connect boards together.

To support dynamic change, all Simics objects are created from classes loaded at run-time. Every Simics model is stored in its own .dll or .so file, and can be loaded at any time. A user does not need to recompile anything to create a new configuration, just load the required modules during a Simics run. Simics is actually fairly similar to the Java and .net environments, in terms of how objects are compiled, loaded, managed and connected. It is not like the static linking model employed in simple C and C++ programs.

One particular advantage of the Simics approach is that each simulation module can be compiled separately, making it very cheap to recompile after a change to the simulated target system.

To manage the complexity of the modelled hardware, Simics allows a virtual platform creator to use a special kind of object known as a component to create logical groupings of hardware units. Components group devices, memories, interconnects, and processor cores together into logical units corresponding to chips, SoCs, ASICs, boards, mezzanine cards, racks, and other hardware units. Components can be reused and arbitrarily nested, modeling any form of hardware design hierarchy. By traversing the component hierarchy, it is easy to understand the structure of the (virtual) hardware system.

Essentially, Simics can model any system. Multiple processors, boards, networks, and heterogeneous computer architectures and target operating systems pose no problem. The largest set-ups used have included up to a thousand processors. The longest simulations have covered many months of target time. The most heterogeneous simulations include dozens of different types of processors from several different families. Multi-core, single-core, symmetric shared memory, asymmetric multiprocessing, 8-bit or 64-bit, Simics has been used to simulate it all.

It is published in [23] that Simics can achieve speeds of up to 1900 MIPS when running single-processor workloads on top of full simulated systems with a real operating system (running an encryption benchmark on a simulated PowerPC 750 target machine with a Linux OS. Host was a 2.4 GHz AMD Athlon64).

2.4 SystemC and Virtual Platforms

Due to their nature as very specialized tools focused mainly on simulation speed, usually Virtual Platforms use different and *ad-hoc* languages to describe their components. We briefly list major Virtual Platforms and the languages they support:

QEMU is a plain C based simulator, and has the possibility to use C and C++ in the description of the devices (or any other programming language suitable to be linked to C).

Simics provides a modular simulation infrastructure where modules representing processors, devices, bus connections, and simulator features can be composed both before and during a simulation run. Modules for Simics can be created in C, C++, Python, or the Simics device modeling language (DML). DML is really a C-code generator that automates the creation of much of the boilerplate code needed to build a device model. Other languages can also be used with Simics, as long as they can link to C-language modules and compile into a DLL or shared object file.

Imperas OVP is based on C language and they offer C and C++ versions of its API to build platforms and simulation models [24]. They also offer a TLM-2 wrapper of any OVP model, allowing the use of the entire platform into a standard SystemC set-up [25].

Synopsys' Innovator is entirely based in SystemC. In fact, this tool is a GUI and a few controls over a standard SystemC simulation set-up. It can use the huge number of SystemC-TLM IPs created by Synopsys.

Cadence framework can use several languages, from Verilog and SystemVerilog to C and C++, and also supporting SystemC models.

Mentor's EDGE SimTest is a Virtual Platform focused on Mentor products like Nucleus Operating System. The entire tool is based on C/C++ language.

Originally, most of the commercial Virtual Platforms don't use SystemC as a description language. Nevertheless, SystemC is a pretty good standard used in industry to describe systems or devices in a very wide range of applications and description levels.

In 2009, different companies started to announce support for SystemC in their tools. Simics did it in July 2009 [26], Imperas released the first versions of its tool in 2008 and started supporting SystemC devices a short time later in February 2009 [27].

The work presented in this dissertation is previous to these announcements, being, perhaps, the first publication of a Virtual Platform with SystemC support. In fact, the Simics SystemC bridge is made by the dissertation author, and other vendors seek his advice for their solutions.

2.5 Checkpointing

Checkpoint save and restore (usually known as *checkpointing*) is a process by which a simulator stores the state of the simulated system to disk, and later loads it back into the simulator, resulting in the exact same simulated system state. For virtual platforms, checkpoints have to include the contents of memories and disks, the state of processors, peripheral devices, and network connections in the virtual system, as well as the state of the simulation kernel including current time and any event queues and simulation scheduler state.

Usually, checkpointing offers these operations:

- Saving simulation state
- Restoring the simulation on the same host machine
- Restoring the simulation on other host machine (other OS, other architecture, etc.)

- Restoring into a bug fixed platform (once a bug in HW is found and solved, the same simulation is continued)

Checkpointing is a key workflow enabler for systems and software development using a virtual platform. With checkpointing, a software developer can save and restore their work at any point and resume it later without having to keep the simulator running. A repetitive simulation procedure such as booting an operating system and loading a set of software applications onto the system can be done once, saved, and then used many times, saving time and ensuring multiple developers have identical system set-ups. This is getting more and more important as simulated systems increase in complexity and workload size. We have seen cases where a system bring-up takes hours, as it involves the simulation of many billions of instructions across hundreds of processors, including reboots and software. Needless to say, in these cases checkpointing is necessary to avoid repeating this [28]. Checkpointing also makes it possible to store a library of booted and configured systems of various forms, for use in regression testing or to try various alternative micro-architectures on the same booted software load.

From the above use cases, different requirements are imposed on the Checkpoint capabilities. The most important are discussed [29]:

- Reliability: The restored simulation must be and behave exactly as the original simulation before the checkpointing.
- Transparency: Ideally, checkpoint mechanism should be applied without modifying existing code.
- Performance: In order to be usable, checkpointing may use short times to do the save and load operation and it should use the minimum data possible to store all information.

- Support for external applications: Virtual Platforms tend to use other applications connected to them, like terminal emulators, GUIs, debuggers, etc. Checkpointing must re-establish connection to these applications.
- Support for O.S. resources: The same applied to O.S, like resources, files, etc. These resources must be managed properly by the Checkpointing mechanism as well.

Checkpointing is commonly offered by VHDL and Verilog simulators like Mentor's ModelSim or Cadence NC-Sim, and some Virtual Platform tools like Simics offer this feature too, but SystemC language was not designed to support this feature, so no implementations of checkpointing for SystemC can be found.

The main goal of this work is to obtain a way to allow checkpointing when using SystemC for modeling parts of a Virtual Platform.

2.6 Checkpointing in Virtual Platforms

Only few Virtual Platforms support native Checkpointing. At time of this dissertation, Wind River Simics[3] [26], CoWare Platform Architect [29] and Cadence Incisive Simulator [30] from the commercial tools support checkpointing in some way.

These tools offer Checkpointing to their users but with different implementations and limitations. Simics implements an explicit checkpointing mechanism, where all devices suitable to be checkpointed must declare their internal attributes and data that it needs to be saved and restored.

3 This support is the final work presented in this dissertation

In the other side, the other tools implement a whole-process save and restore, meaning that the save operation is a memory-dump of the simulation process.

In the open-sourced Virtual Platforms, QEMU supports saving and restoring the state of the simulation in a very similar way to commercial checkpointing, using explicit checkpointing (i.e. devices explicitly declare their internal attributes).

3 Virtual Platforms and SystemC

In this chapter we introduce the work related with this PhD on adding SystemC to two different Virtual Platforms.

This work (with the different solutions proposed) was the seed of all later work related in my PhD thesis.

The first sections introduce QEMU, which is the simulator I choose to start the research on Virtual Platforms and SystemC. Later in this chapter, work done with SystemC and QEMU is introduced, and how this same philosophy was applied to the commercial tool Simics.

To end the chapter, I'll introduce another strategy to add SystemC capabilities to QEMU simulator. This project was named QBox, and it was released as a patch to QEMU and used in another commercial tool.

3.1 QEMU-SC

QEMU-SC is the name of the first solution for adding SystemC devices to QEMU simulator.

3.1.1 Motivation

Since QEMU is an open sourced Virtual Platform, and seeing the lack of Virtual Platforms supporting SystemC as a modeling language, the idea of adding this support to QEMU appears.

First versions of this work were based on RTL instead of a higher level of abstraction, emulating at a very low level a PCI controller to interface with a single SystemC device [31].

This initial version presents the known problem of RTL simulations, that is the simulation speed. In spite of this, we simulated and tested some large systems, including a MPEG-2 decoder based on a SoC with some accelerator described in SystemC, and a NoC MPEG-4 encoder described entirely in SystemC and presented to the CPU as a special device connected to fabric bus [32; 33].

Those first steps were evolved into the actual QEMU-SC project, when we did detect the need of raising the abstraction level to increase simulation speed.

We raised the abstraction level to TLM and allowed multiple SystemC devices to be plugged into QEMU and this enhanced version is what we explain in the next sections, because the RTL version was discontinued some time ago.

3.1.2 Architecture

To give connection to a SystemC device into QEMU, the first step is to have a QEMU module capable of publishing itself to the Virtual Platform with the same characteristics as the SystemC device. This module is called sc_link, and mimics the configuration space (in case of PCI devices) or memory map (in other buses like AMBA) and receives data reads and writes. These accesses must be passed to the SystemC device, and the response from the device sent back to QEMU. This is done through the SystemC bridge, which also manages synchronization time between simulators.

The SystemC bridge (sc_bridge.cc) is called by the SystemC link when an access to the SystemC devices is made by the CPU emulator. The SystemC bridge will act as a TLM-2 Initiator for the transactions, and the SystemC device as the Target. Once called, the bridge creates and fills the TLM-2 transaction and performs the transaction operation, managing SystemC simulation time to finish the transaction. The block diagram is shown in *Figure 3.1*, with SystemC blocks in yellow.

With this partitioning, adding SystemC support to another QEMU virtual platform is as simple as writing a new sc_link module for that specific platform (and its specific bus fabric). The sc_link will do the conversion between that platform's bus access mechanism and the data necessary to call the SystemC bridge functions (basically, address and data) as depicted in *Figure 3.2*. The SystemC bridge module will actually construct and send the TLM-2 transaction.

For most bus structures it is easy to extract the right data to create transactions, however, when the platform is x86 based, the data that our SystemC link receives is the virtual address for the PCI device. As PCI addresses are managed by the BIOS or the O.S., we cannot know in advance what the address of our device is. This calculation is performed in the SystemC link, so the bridge always receives the operation data in the form of address and data.

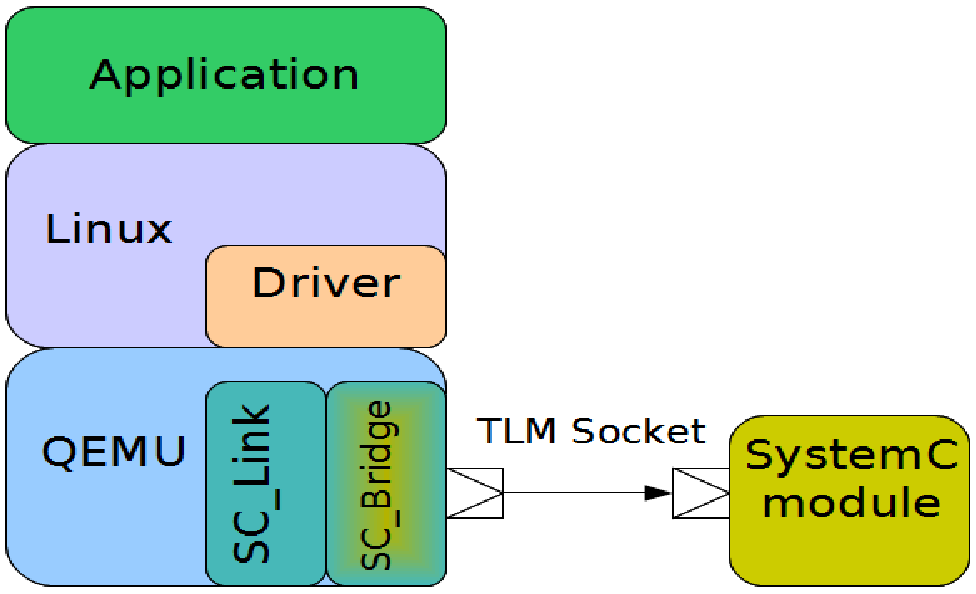


Figure 3.1: Block Diagram (SystemC blocks grayed)

This partition gives us flexibility to port our work to other supported virtual platforms in QEMU, only needing to write a new sc_link module for the desired platform, and re-using the bridge module for all platforms.

We compile together both the SystemC bridge and the device to get a single object file with the SystemC device, the SystemC bridge and the OSCI simulator. This object file is linked with the rest of the QEMU to obtain the QEMU executable.

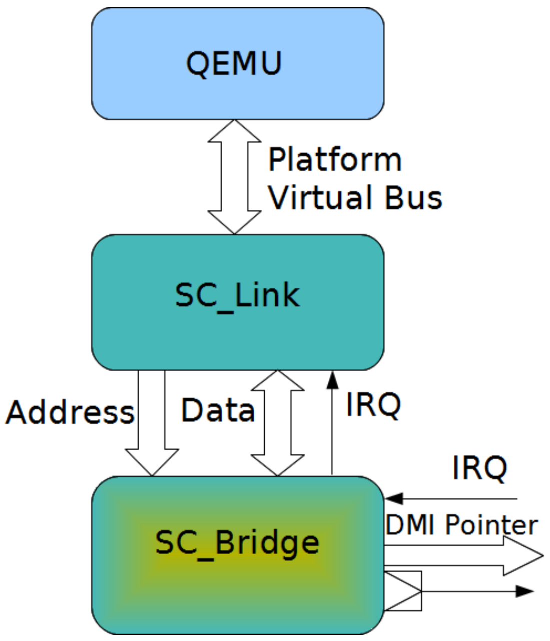


Figure 3.2: QEMU – SystemC Dataflow

3.1.2.1 Changes to SystemC main

As we put the entire SystemC subsystem as a slave of the QEMU, including the OSCI kernel, the *sc_main()* function is not used in the simulation. This way, the *main()* function belongs to fabric QEMU code. Current OSCI kernel implementation invokes module callback functions[4] when *sc_start()* is called for the first time, so no side effect is observed when we remove *sc_main()*.

In case the previous system has a large and complex *sc_main()*, all this code should be moved to the initialization function of the top level module of the SystemC sub-system.

This top-level module will be called by QEMU-SC before starting the simulation. This change should be very straightforward and quite simple to do.

3.1.3 Implementation

In this section we explain details of the implementation of the previously commented solution to add SystemC capabilities to the QEMU simulator.

3.1.3.1 Synchronization

The main problem when joining two simulators (like QEMU and SystemC) together is the synchronization between the two notions of time that exist in the two simulators.

A possible solution would be to synchronize SystemC with QEMU every system clock cycle, but it would suffer a great performance penalty.

Our initial approach is based on a lazy connection, and the notion of 'quantum time' introduced by OSCI's TLM-2 standard. We synchronize both worlds only when it is necessary to do so. In the simplest case, we could imagine

4 *before_end_of_elaboration(), end_of_elaboration(), start_of_simulation(), ...*

that SystemC is used to describe a small number of devices of the virtual platform. Hence, we could assume that the majority of simulation time is spent in the QEMU CPU emulator rather than in the devices. Thus, we could have SystemC "frozen", and "woken up" only when the subsystem is accessed. This is a simple (and insufficient) case.

We assume all SystemC models are well behaved and adhere to the TLM-2 standard. Hence they should provide both LT and AT interfaces (both blocking and non blocking). We make use of the LT interface (blocking) such that the model can advance SystemC time in order to complete the transaction. However, this is not sufficient. Although system devices may respond immediately to an access to the register file, they may (meanwhile) start to perform some other computation in the background.

This is the crux of the synchronization problem.

To take this into account (i.e. the case that a transaction sent to a device triggers some event in the future in the same device), every time a transaction is processed, the SystemC bridge needs to know for the next event in the SystemC simulation.

To be able to ask to the OSCI kernel for any of its internal characteristics, we need to very lightly modify the kernel core. We just need to add our bridge as a *friend* class of the *sc_simcontext* class of the OSCI kernel. That *sc_simcontext* class is the principal simulator module of the OSCI simulator. In this way, our bridge can poke for the events list inside the simulator or any other information it would need.

If a future event exists, the corresponding event time is posted in a QEMU event queue in order to trigger the bridge again to perform a synchronization. When the QEMU time equals the event time in SystemC simulation, and the bridge is triggered, the SystemC bridge allows the SystemC simulation to continue (calling *start()* again), allowing it to process the event that had been scheduled. Again, when this has been done, the bridge checks again to see if there are new outstanding SystemC events.

For sanity, and also to adhere to the notion of 'quantum' introduced by TLM-2, at each 'global quantum' (as registered in the quantum keeper) we also synchronize both worlds. This synchronization may only update the SystemC time to be the same as QEMU time, and because we are assured that there are no pending events in the SystemC simulator, the time update operation is very simple and fast, not penalizing the performance of the entire simulation. However, this also helps to ensure that external inputs into SystemC models are dealt with.

In some cases, this is not sufficient. In the case of devices that have external I/O, because the device is "frozen" most of the time, the I/O operations may need to be handled by the SystemC bridge in order to gain enough responsiveness to the external input. This can be arranged notifying the SystemC bridge when an external event is happening, so that it can manage the SystemC simulator correctly.

```
function syncronize() {
      if (qemu_time before systemc_time)
            stallQEMU(systemc_time – qemu_time)

      else if (qemu_time ahead of systemc_time) or
                  (pending_event)
            sc_start(qemu_time – systemc_time)

      if (next_systemc_event = 0) or
            (next_quantum before next_systemc_event)
            Synchronize(next_quantum);
      else
            Synchronize(next_systemc_event);
}
```

Figure 3.3: Synchronization pseudo-code

In other words, the mechanism responsible for connecting the ‘outside world’ to the SystemC model must ‘notify’ the SystemC bridge (using a function call) when new input is given to the model. The SystemC bridge will then arrange to synchronize time again, and ‘run’ the SystemC kernel. This mechanism is important, and has been demonstrated using a UART connected to a terminal window.

Figure 3.3 shows pseudo-code for the synchronization function that is called at every transaction to the SystemC device.

3.1.3.2 TLM-2 socket

Communication between the bridge and the SystemC device is done through a TLM-2 socket. We use the GreenSocs Generic Protocol Socket [34]. This socket allows connection of OSCI TLM-2 base protocol devices and, at the same time, simplifies the data management and the user code of the model. It also allows us to connect this socket to multiple targets, allowing us to plug more than one SystemC device into the same QEMU ‘socket’. The GSGPSocket belongs to the GreenSocket family, which also includes a very similar socket (that is interchangeable with it) that emulates the PCIe bus, allowing us to model PCIe with more detail. Other similar GreenSocket based sockets which encapsulate other protocols (AMBA, OCP, ...) could be used in our SystemC bridge without any significant work.

For efficiency, a single transaction object is used on all simulations, and is built in the QEMU initialization phase. This transaction object is reused for all accesses during the entire simulation.

This transaction is filled with the required data for each access to the device. This involves filling the fields corresponding to access type (Read or Write), the address to be accessed, and the data to write (in case of write access). The rest of the transaction is static throughout the simulation.

3.1.3.3 DMI

If the system bus in the QEMU emulation is capable of identifying Memory transactions (as the PCI bus does), our SystemC link tries to use DMI functions when the CPU accesses a memory region of the device. The SystemC bridge uses the DMI capabilities of a SystemC device if the SystemC device has them, and the memory region is accessible through DMI. If it is not possible to access using the DMI, the SystemC bridge uses normal transaction to perform the access.

Also, TLM-2 Targets can notify the Initiator that it is able to support DMI using the DMI hint attribute in the transaction payload.

We implemented DMI mechanism for the case of the x86 based platform and PCI devices registering themselves as a Memory devices (a PCI device can publish itself as I/O region or Memory region) or devices that signal using DMI hint that they can be accessed through DMI.

3.1.4 Results

In order to validate our approach and solution, we build different test-benches, involving for each test:

- the SystemC device
- O.S. Driver
- test application running on QEMU virtual platform

All tests were done using an unmodified Debian GNU/Linux running as a guest on QEMU and we focused on only two of the available QEMU platforms: intel x86 PC and ARM's VersatilePB [8].

3.1.4.1 Simple experiment

We started with a simple SystemC device that publishes a register file. This register file is accessed through the TLM-2 socket. We mapped this register file directly to the AMBA bus in the virtual ARM platform and we mapped the same SystemC device to a simple PCI device with a single Memory BAR for the virtual PC x86 platform.

For both platforms, we wrote the corresponding Linux driver that enables the SystemC device as a character device, allowing user applications to read and write to the device as a file system and the driver sending the operation to the virtual platform system bus.

With this initial set-up we could check that all components were running as expected, and that both simulators were properly synchronized.

3.1.4.2 Interrupt generator experiment

The next experiment was adding the capability of generating an interrupt to the SystemC device when one of its registers was written to. Once triggered, the interrupt generation mechanism in the device will generate interrupts to the system for a periodic time (for this experiment we set this time to 10 seconds). The interrupt generation can be disabled with a read to the same register. The way the SystemC device generates this periodic interrupt is simply by posting a SystemC event in the future, and having a SC_METHOD sensitive to that event that triggers the interrupt and posts the event again (*Figure 3.4*).

This second experiment demonstrates the management of the event queue from the OSCI simulator, and the success of our strategy of having the SystemC simulator frozen and running only when there are events pending.

```
...
SC_METHOD(irq_generation);
sensitive << irq_event;
dont_initialize();
...
```

(a) constructor

```
void slave_dummy::irq_generation()
{
     irq_event.notify(10, SC_SEC);
     // triggers a Interrupt
     irq_line = true;
}
```

(b) SC_METHOD

Figure 3.4: IRQ generation code (a) constructor (b) SC_METHOD

3.1.4.3 Testing DMI

Another experiment was designed to test DMI accesses. We add DMI capabilities to our SystemC device. This functionality is transparent to the O.S., driver and user application. This enhancement should give better performance (when accessing memory devices) than simply using standard transactions across the Sockets.

The results of this last experiment didn't show a performance enhancement for the DMI mechanism. We think this is due to the fact that QEMU accesses to devices using single word accesses instead of using bus bursts, disabling the theoretical improvement of direct access to memory, that will boost performance when used to access large chunks of data at once.

The resulting device and set-up is shown in *Figure 3.5*. Notice we developed the same tests for the ARM platform and for the x86 platform. To perform that change it is only necessary to modify the sc_link module for each different platform, reusing all other modules.

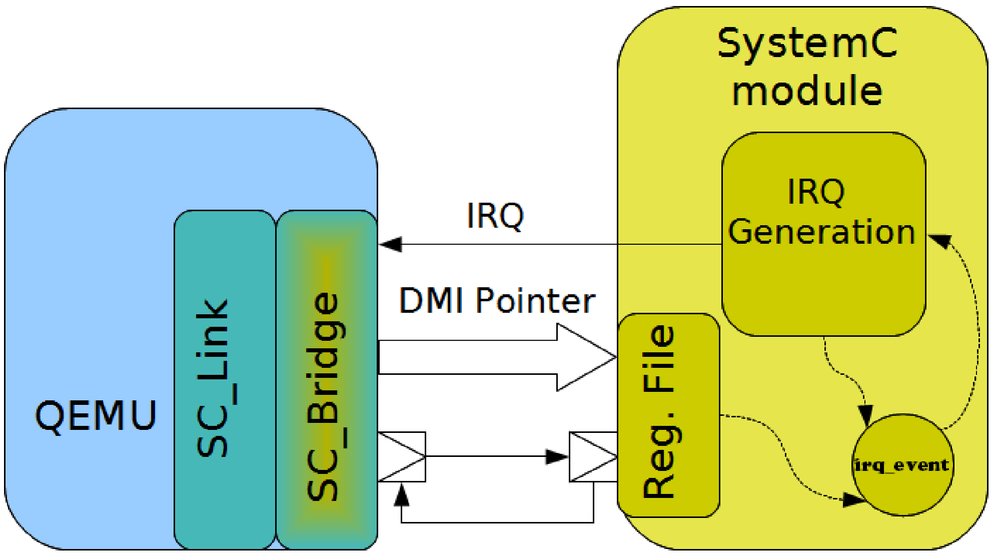


Figure 3.5: Final test bench module

3.1.4.4 MPEG-2 decoder

The last experiment was the simulation of an ARM based SoC device with HW accelerators for MPEG-2 decompression. We used the decoder implementation and source code from MPEG Software Simulation Group [35].

We added to the system a HW module for the Inverse Discrete Cosine Transform (iDCT) calculations of the decoder [36]. This implementation performs 11 multiplications and 29 adds per 8 point 1D iDCT. We wrote two different modules, one for columns calculation and one for rows calculation. These devices are attached to the system bus, and once they get the 8 inputs needed, they perform the calculations and the results can be read back to the processor.

We tested the system with 8 different implementations:

- Without accelerator module.
- Without acceleration, but with the modifications to access an accelerator module (write and read to special file system device)
- With one (columns) accelerator module written in QEMU plain C style.
- With one (columns) accelerator module written in SystemC.
- With two (both columns and rows) accelerator modules written in QEMU plain C style.
- With two (columns and rows) accelerator modules written in SystemC.
- Same as the above, but without the computation code inside the modeled accelerators

As shown in Table 3.1, adding only a module described in SystemC penalizes the global simulation time by about 12% in relation to the same module described in QEMU plain C style.

The differences between A and B implementation is the addition of the infrastructure to use HW accelerators like drivers, accesses to devices, etc.

The first observation to make is that simulating the same device in SystemC (implementation D) instead of using QEMU native C language (implementation C) introduces a penalty of about 13%. We tried to identify the reason for this penalty, although this number seems to be good enough for our platform.

As observed in implementations E and F, the penalty is only about 11% when using two SystemC modules instead of the previous 13% for only one module (C and D implementations). This could indicate that the main performance penalty is because of synchronization and communication between the two simulators.

Table 3.1: Execution time for different MPEG-2 decoder implementations

Name	Implementation	Execution time (s)
A	Without accelerators	1.10
B	Without acceleration w/ drivers	22.10
C	QEMU style1 accelerator	19.83
D	SystemC 1 accelerator	26.65
E	QEMU style2 accelerators	28.13
F	SystemC 2 accelerators	32.17
G	SystemC skeleton (1 acc.)	25.50
H	SystemC skeleton (2 acc.)	27.29

To prove that the main penalty is there, we removed the computation part of the SystemC modules leaving only the communication part, and ran the tests again. As shown in results for G and H implementations, the main performance drop is due to those two mechanisms instead of the simulation of the modules. The simulation of the modules themselves is 6% to 15% of the total time.

Comparing G to B implementations, we observe that the total time used in synchronizing and communicating both simulators is about 11% of the time. That percentage is consistent with the other results in the same table.

3.2 Virtutech Simics Bridge

While QEMU provides an interesting Virtual Platform example, to prove that our solution (and related methodology) is widely applicable we have also added SystemC capabilities to Virtutech's Simics[5] [12]. This virtual platform simulator is based on a set of devices which are pluggable into a CPU emulator together with a simulator kernel that manages events and transactions between devices and the CPU.

Simics supports devices written in many programming languages including C++, C, DML and Python. These modules may be written accordingly to the Simics API that allows plugging any type of device into the Virtual Platform.

We built a new module acting as a bridge between Simics and any SystemC written device. That work involved managing OSCI kernel from an other simulator, being the SystemC kernel a slave of the Simics kernel that acts as Master of the entire simulation.

5 Actually this product is named Wind River Simics.

3.2.1 Architecture

We have developed a device to be plugged into Simics version 4.0 (in the same way we described the device for QEMU) to enable the use of SystemC with Simics. Due to the Simics layered schema, we didn't need to use a QEMU sc_link equivalent: Simics devices only receive transactions that are relevant to themselves, regardless of the system bus of the emulated platform. Thus, we only need to use the SystemC Bridge module for the integration as shown in *Figure 3.6*.

What we did was write a device for Simics, using its API and interfaces acting as a bridge between the SystemC device and Simics. In addition, we embedded the OSCI kernel in the module itself, linking the module with the SystemC library, being a single object from the Simics point of view.

This bridge can use all mechanisms and functions from Simics API, and it is responsible to interface between all Simics I/O like serial links.

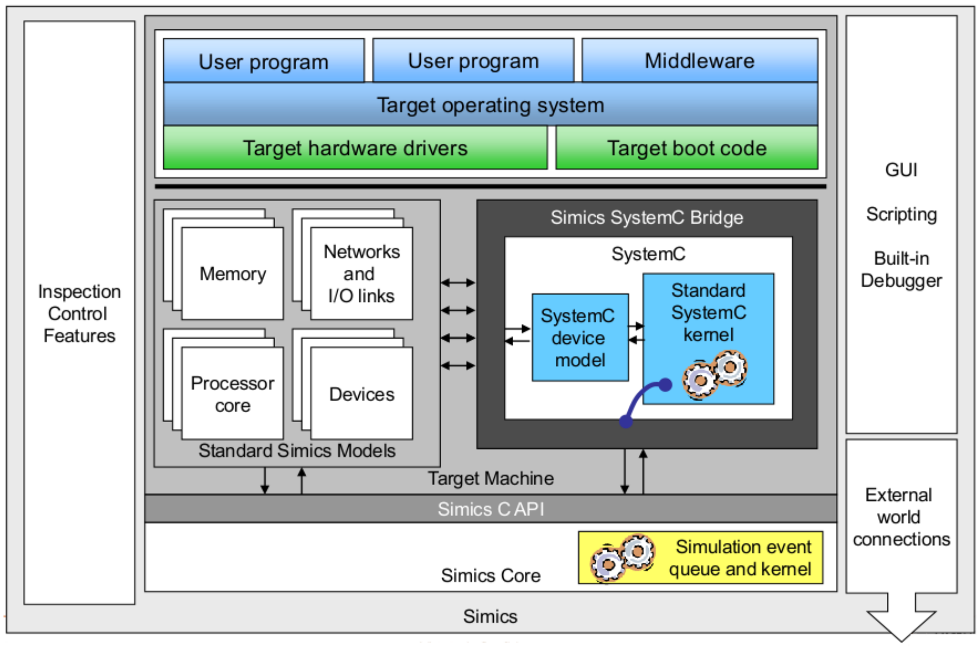


Figure 3.6: Simics and SystemC Bridge (source WindRiver)

3.2.1.1 Changes to SystemC main

In this implementation we use the same mechanism that we used for QEMU-SC, as explained in *3.1.2.1*.

In the case of Simics, every device has an initialization function (which is called when this module is instantiate) to be used in the simulation. This *init* function can be used for this purpose directly (as the *sc_main* of the SystemC simulation).

3.2.2 Implementation

The bridge between two simulators is in charge of two related tasks: adapt transactions and data moving between both simulators; and manage simulation time between the two simulators.

3.2.2.1 Simics Data Transfers

In Simics, every call to a device model is synchronous. This means that the data from the operation is returned immediately, with no notion of time in it. Basically, it is a function call in any of the multiple programming languages that Simics support.

This communication mechanism fits perfectly with OSCI TLM-2 blocking calls in Untimed (UT) or Loosely Timed (LT) calling style. Thus, our Simics bridge should work with this type of TLM communication.

But existing TLM-2 based models can also use Approximately-Timed (AT) style. So in order to re-use them in Simics, we add this style to our Bridge. In this case, although the communication between the Bridge and the TLM-2 model is using non-blocking calls, the bridge may finish all the phases of the communication before returning it to Simics as shown in *Figure 3.7*.

In this case, our bridge runs SystemC simulation time ahead of Simics time to allow the AT device to finish its protocol phases. This ahead time is passed back to Simics as the response time to the device, and it is taken into account by Simics when managing its own simulation time. In this way, at the end of the transaction, both simulations are rightly synchronized again.

3.2.2.2 Synchronization

We use the same synchronization management as described in the previous section *3.1.3.1* for the QEMU-SC Bridge.

Temporal decoupling mechanism is also supported by this bridge, being configurable by the user (at configuration phase) the amount of time (quantum time) that the SystemC device can run ahead of current simulation time.

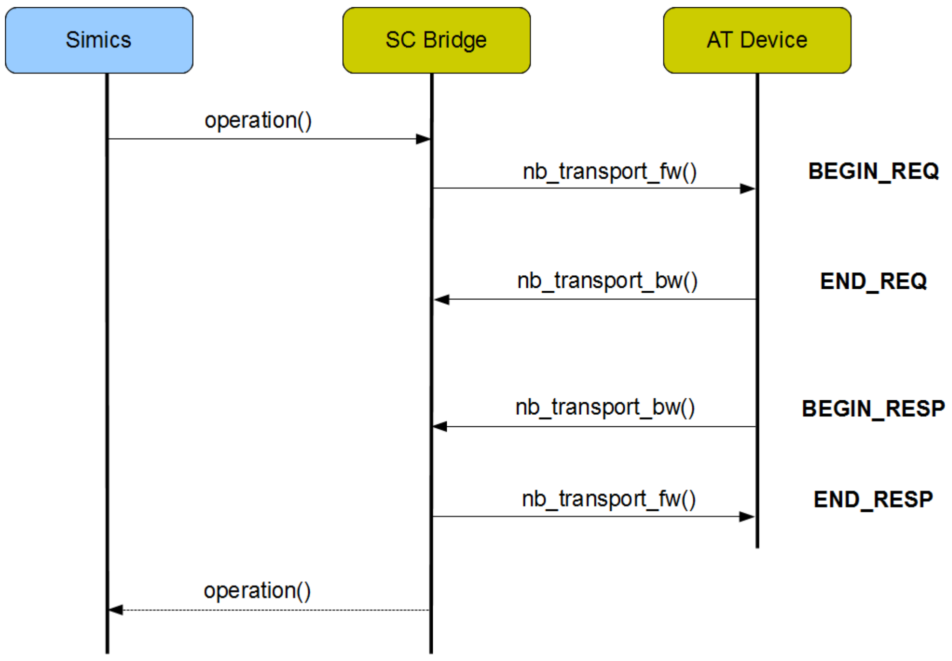


Figure 3.7: Communication schema between Simics and an AT device

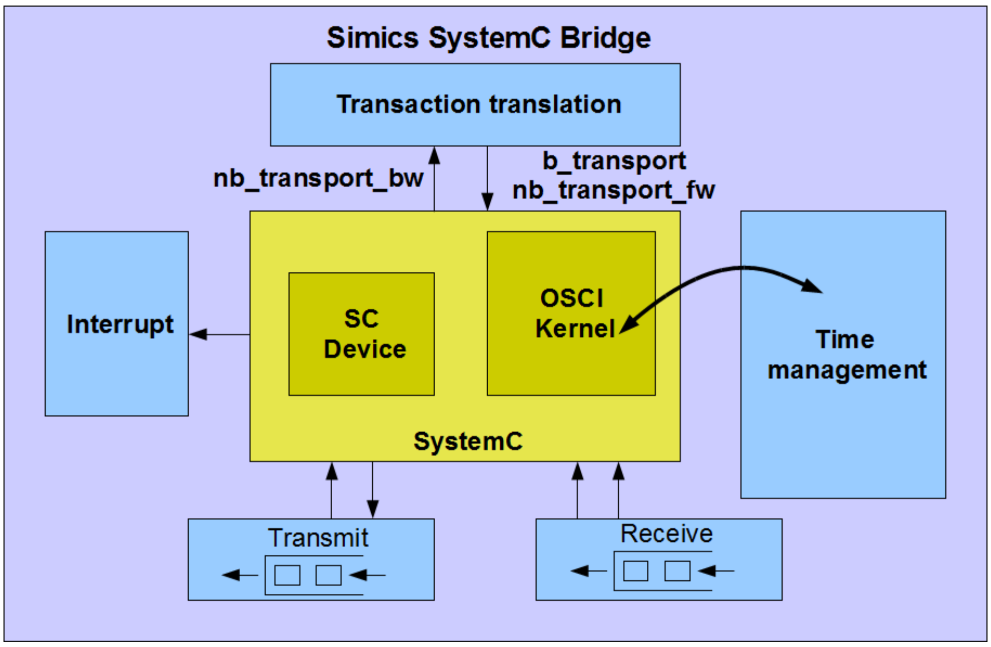


Figure 3.8: Simics bridge block diagram

3.2.2.3 Checkpoint

This work also included adding checkpointing to SystemC (i.e. the ability to stop and store to disk the whole simulator state so that it could be resumed at a later date). We achieved partial results, enabling checkpointing for restricted cases, and leaving the responsibility to the user to identify the device variables that are crucial in order for the device to be properly saved and restored. More details of this work can be found in the next chapter.

3.2.3 Results

We provided to Virtutech a full compatible SystemC TLM-2 Bridge module for Simics 4.0. and we ran several tests to ensure the correctness of the module and its functions. This module has the structure depicted in *Figure 3.8*.

We tested the integration using a SystemC NS16550 UART [37] plugged into the Simics simulator. The UART device was written from scratch in TLM-2 LT/AT style, with no temporal decoupling support.

The virtual platform used in this test was an AMCC “ebony” board [38], based on a PowerPC 440GP SoC. This board was running full Linux kernel and a synthetic test for the UART device.

The integration was successful, with an imperceptible impact of using the SystemC UART instead of the simple UART provided by Simics in normal use.

When using a micro-benchmark program that pushed characters to the UART as quickly as possible, the performance was reduced to about 5% when using the SystemC UART instead of the Simics standard UART model. Results are summarized in *Table 3.2*.

This 5% penalty is caused by the data translation between Simics and SystemC and the overhead of synchronization, demonstrating that the SystemC bridge does not present a significant performance problem.

Virtutech announced the availability of checkpointing and the bridge for SystemC devices on July 2009 [26] and since then that feature is present in subsequent Simics versions (4.2 and 4.4).

Table 3.2: Results for UART experiments on Simics

	Cycles	**MIPS**
SystemC UART	1944.1625	0.26
Simics UART	1864.265	0.27

On February 2010, Wind River announced the acquisition of Virtutech [39] and the addition of all Virtutech products (mainly Simics Virtual Platform) to the Wind River embedded software product portfolio. The Simics SystemC bridge is in its product portfolio [40].

3.3 QBox

Another way to use the QEMU simulator in SystemC is to wrap it to be a SystemC standard module and get a standard simulation schema with the QEMU acting as an Initiator and its devices as Targets (as depicted in *Figure 3.9*).

To achieve that, we need to remove parts of the SystemC bridge related to managing the SystemC simulation (because now QEMU will act as a standard SystemC module) and modify QEMU to allow to to pause and resume it in order to be able to yield control (i.e. call *wait*()).

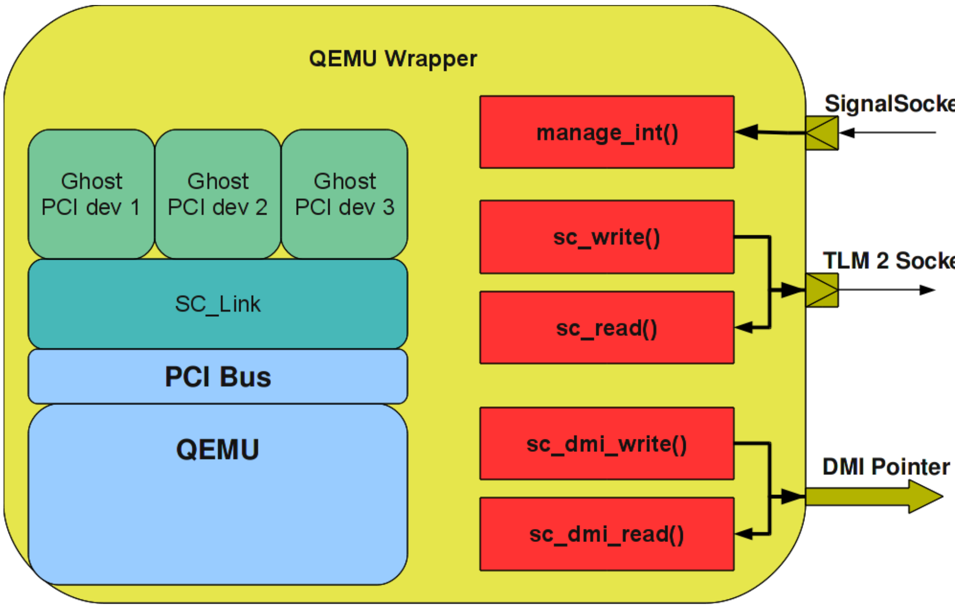


Figure 3.9: Wrapped QEMU block diagram

The use of the same TLM-2 sockets in both implementations, allows the use of the same SystemC devices in both simulations.

3.3.1 Architecture

The original QEMU is based on an endless loop that performs all the necessary steps to do the simulation. We broke this loop with a call to a synchronization function that computes the difference between QEMU time and SystemC time. If SystemC time is far enough ahead of QEMU time, the wrapper yields control to the simulation kernel[6].

Our wrapper contains an SC_THREAD that runs the QEMU modified main loop and yields to the simulator when necessary. Also, the wrapper has two functions (sc_read/sc_write) to allow QEMU to use them to access SystemC devices when requested. These functions (sc_read/sc_write) use a TLM-2 socket in the same way we explained in section *3.1.3*.

The wrapper works together with a special **sc_link** that publishes itself to the QEMU system bus like it would the SystemC devices. The sc_link device captures all data accessed from QEMU to those ghost system bus devices and sends the data to the proper SystemC device.

In the case of an x86 system using PCI bus, this sc_link reads a configuration file that informs on the location of the PCI configuration register for all SystemC devices, and it publishes these devices to PCI QEMU subsystem. Then, when QEMU is running, every access to those PCI devices will become a transaction to the proper SystemC device through our wrapper.

The same strategy can be applied to any of the platforms and system bus that QEMU supports.

6 The amount of different time between both is controlled by the user.

3.3.2 Implementation

We wrote a TLM-2 Initiator module named *qemuWPSC* that wraps QEMU and becomes the module to use in simulations willing to use QBox. Below we detail different aspects of the implementation of this wrapper.

3.3.2.1 Synchronization

The implementation of the QBox includes modifications to the QEMU kernel source. The wrapper calls QEMU's main loop once SystemC simulation is started, and the QEMU main loop calls a check function to yield control, if necessary, to the simulation kernel.

The wrapper is designed to use Temporal Decoupling, so the synchronization function uses TLM-2 Quantum Keeper to manage its own simulation time and only yields control to the simulator when the time by which QEMU is ahead is greater than the global quantum allowed in SystemC simulation time.

When the QEMU wrapper must yield control back to the simulator, it freezes QEMU simulation and then yields to the SystemC simulator. Once control comes back to the wrapper, QEMU execution is resumed again.

3.3.2.2 QEMU Data Transfers

When QEMU accesses the ghost device plugged into the system bus, it calls a callback function in sc_link previously registered. This function captures information regarding the access type (read or write, exclusive access, etc.), address and data (in case of a writing). All these data accesses are then passed to the wrapper which builds the corresponding transaction. Once built, the transaction is sent through the TLM-2 Socket to the SystemC device using blocking mechanism (Loosely Timed).

As this mechanism allows Targets to yield control to the simulator (calling *wait()*), the wrapper suspends QEMU execution before data communication begins. In this manner, we ensure that QEMU is stopped in case one Target yields to the SystemC simulator.

Once the blocking transport returns, the wrapper breaks the transaction up into system bus data and sends it back to sc_link, and resumes QEMU execution again.

Because bus accesses in QEMU are synchronous, QEMU expects a bus access to be complete once callback function returns. If our Wrapper needs to use AT mechanism (non-blocking) it might manage all phases until completion of the transaction before that. This mechanism is the same we used for the Simics bridge as explained in *3.2.1* and depicted in *Figure 3.7* in page 72.

3.3.2.3 Using QBox standalone

The wrapper we wrote transforms QEMU to a normal TLM-2 SystemC module, no matter what kind of mechanisms or code is inside QEMU code.

We can even build QEMU in a static library way, and use it with our wrapper and link together with the rest of SystemC modules to obtain a perfectly valid SystemC simulation executable. Using this mechanism, we are able to give third parties a binary[7] file with QEMU in it.

7 A static linked library file in Linux and gcc.

3.3.3 Results

We have built the same testbench as we did for the first solution (QEMU-SC), with the difference being that now we have two SystemC modules (the QEMU Wrapper and the SystemC device) in the same hierarchical level (which are connected) and a top level module that instantiates the two modules and binds the sockets together.

3.3.3.1 Simple system

Our tests with the basic setup described in the previous section don't show any performance difference with the previous proposed solution.

3.3.3.2 Complex system

To demonstrate the power of that configuration, we built a more complex system, involving two network devices. This system is an Intel PC x86 platform on the QEMU side and we attach two instances of an Ethernet module described in SystemC as seen in *Figure 3.10*.

This module is emulating a ne2000 Ethernet PC card by National Semiconductor based on DP8390 Ethernet chip [41]. This module was written from scratch based on the same device in QEMU source code.

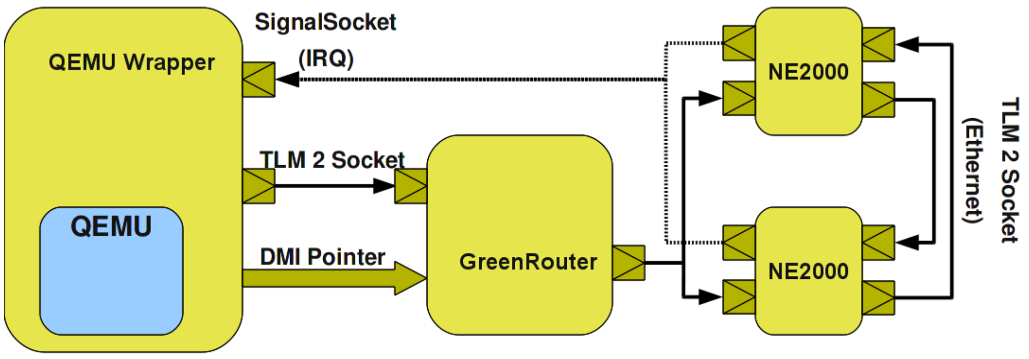


Figure 3.10: Complex test bench block diagram

The SystemC ne2000 module has a TLM-2 socket (GSGPSocket) to interface to QEMU and one Initiator and one Target port to connect to Ethernet side. Also, it has another socket (signal socket) to manage the interrupt interface to CPU.

We connected both Ethernet sockets between the two modules, simulating that the two Ethernet cards are linked together using a cross-over cable or a standard Ethernet switch.

These two devices are connected to QEMU wrapper using a Router (GreenRouter) by GreenSocs [42]. The router has the same memory map used by the sc_link, and routes the transactions to the correct device according to that memory map.

These two devices appears to QEMU as two regular PCI devices, and the O.S. detects them and configures them as it would do in a real platform. With the connection between them shown, it is possible to ping one card from the other, send data between the two cards, etc., thus validating the correctness of all modules designed.

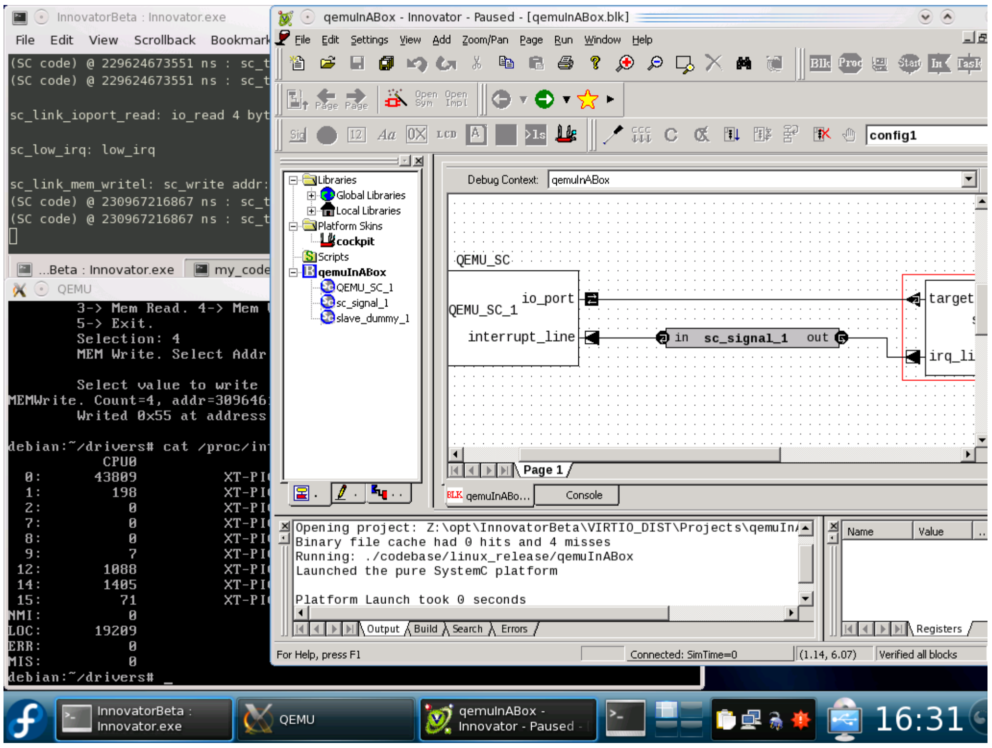
3.3.3.3 Using QBox in Innovator

This QBox module allows us to use QEMU in any SystemC set-up, including complex simulation tools like Innovator by Synopsys [17] as shown in *Figure 3.11.*

To use QBox with Innovator, one restriction was to use it in binary form, so we provided a static library with the modified QEMU and the code for the Wrapper. Within Innovator, we were able to import QBox and our test modules, and prepare and run the simulation using the tool without any change in the tool or our design [43].

The integration of QBox with Innovator was presented by GreenSocs at DAC 2009 in the “Synopsys Standards Booth Theater - GreenSocs Interoperable Modeling Kits Working with Innovator” [44].

With this test we demonstrate that the QBox solution becomes a truly standard SystemC module and it is possible to use it in any of the new ESL tools that recently appeared.



```
(SC code) @ 229624673551 ns : sc_t
(SC code) @ 229624673551 ns : sc_t

sc_link_ioport_read: io_read 4 byt

sc_low_irq: low_irq

sc_link_mem_writel: sc_write addr:
(SC code) @ 230967216867 ns : sc_t
(SC code) @ 230967216867 ns : sc_t
```

```
        3-> Mem Read. 4-> Mem
        5-> Exit.
        Selection: 4
        MEM Write. Select Addr

        Select value to write
MEMWrite. Count=4, addr=309646
        Writed 0x55 at address

debian:~/drivers# cat /proc/in
           CPU0
  0:      43809          XT-PI
  1:        198          XT-PI
  2:          0          XT-PI
  7:          0          XT-PI
  8:          0          XT-PI
  9:          7          XT-PI
 12:       1088          XT-PI
 14:       1405          XT-PI
 15:         71          XT-PI
NMI:          0
LOC:      19209
ERR:          0
MIS:          0
debian:~/drivers# _
```

```
Opening project: Z:\opt\InnovatorBeta\VIRTIO_DIST\Projects\qemuIn
Binary file cache had 0 hits and 4 misses
Running: ./codebase/linux_release/qemuInABox
Launched the pure SystemC platform

Platform Launch took 0 seconds
```

Figure 3.11: QBox in Innovator

4 Checkpointing for SystemC & TLM-2

This chapter introduces our work in adding Checkpointing features to SystemC language. We start with our motivation to do that job and the status of the different attempts to offer a solution to that necessity. Finally, we detail our solution and implementation as well as various tests and the results we get from them.

4.1 Motivation

If we want to use SystemC and TLM-2 to model parts of a Virtual Platform, a key feature we need is to make those models checkpointable. Current SystemC implementations do not support checkpoint, and because of the C++ inherent in SystemC, adding this feature seems a tough job.

This chapter describes our proposal and our progress in this field, that ended by achieving a good solution and implementation. Our approach consists firstly of inspecting the SystemC kernel to access its state variables; then giving a mechanism to designers to explicitly publish its state variables; and finally adding a checkpoint feature to SystemC primitive channels.

4.2 Related Work

SimOS and IBM Mambo full-system simulator [45; 46] use a checkpointing mechanism quite similar to what we propose, with explicit model state separate from the implementation state. Typical uses for checkpoints in Mambo are to save complex system setups and to switch between different implementations and different abstraction levels.

The Boost C++ serialization library can save and restore the state of C++ objects in a program [47]. It provides portability and upgradability to new versions of the code. However, using this library with SystemC would have required a rewrite of the SystemC kernel to use serialized objects to store the whole state, and it is not clear how this would work with the cooperative multitasking nature of the SystemC kernel. The Boost library uses the same vision as our solution on the state: only data stored in C++ object members are serialized, neither temporary values on the stack, nor the state of threads in the system. Also, using Boost would require larger modifications to existing device models than our solution.

Another alternative solution that has been proposed is to save the entire contents of the memory of a running simulation process. At least in theory, this should work for any code, without modification, and including thread state. This is what it appears that CoWare and Cadence are doing in their recently announced checkpoint support for SystemC, even if public details are quite scarce [29; 30; 48].

The memory-dump solution has several limitations compared to our approach. As a checkpoint contains the state of the stack and heap, it is tied to the particular data layout and stack-frame layout of the code the simulation started with. Thus, it cannot be restored on a different machine (even a minor change such as a Linux kernel version or a different set of system libraries can break it), nor can it be used with an updated or different model code.

In [29] a strategy is presented to release external resources when starting the checkpointing phase and reconnect them again when restoring the simulation. These resources can be O.S. resources like temporal files or accesses to real devices (serial devices, etc.), debuggers or GUIs. As their strategy for checkpointing is to save the whole process, they need to release and restore these resources explicitly each time a checkpoint is taken. With these particularities, the solution presented has a severe performance penalty (more than 2 seconds to do a checkpoint and more than 6 seconds to restore it) disabling all chances to be useful for reversal simulation (where several checkpoints are taken in order to restore them in reverse order). Also, the size of the checkpoint data will be very large, in order of hundreds or thousands of Megabytes.

An even more heavy-weight solution is to place the simulation inside a VMware virtual machine, and use the whole-machine snapshot function of VMware to save and restore the simulation. This works, but the designer cannot change the code, and the size of the snapshot is the size of the total memory of the virtual machine, which is usually on the order of 1 GB (or more).

4.3 Design and Implementation

To support checkpointing in SystemC (and indeed in any simulator system), there are three problems that have to be solved:

- Saving and restoring the simulation state of all models in a simulation. Model properties like register contents, current states of state machines and similar, must all be saved.
- Saving and restoring the simulation kernel state, such as event queues, FIFO values, mutex state and the current simulation time.
- Saving and restoring the simulation configuration in terms of which simulation models form part of a virtual platform, and how they are connected.

We have addressed these problems for device models written in SystemC, using the OSCI reference SystemC simulator version 2.2.0. To create a complete system involving processors, memories, and software, we used Virtutech Simics (and QEMU[8] for some experiments) to provide the rest of the platform, as well as a proven checkpointing infrastructure[9]. That allowed us to focus on the core technology issues rather than checkpoint file structure and disk I/O.

4.3.1 Model requirements

Checkpointing requires a model to explicitly define the state that will be saved and restored. In our implementation, we have used the GreenControl [49] parameter mechanism to simplify the implementation in SystemC and make the changes to the SystemC code as small as possible.

GreenControl lets us mark variables in a SystemC module as managed parameters using the gs_param<> template. From a SystemC code point of view, such variables can be used just like any other variable, with the added benefit that its value can be accessed and changed from the GreenControl system outside the module itself. The GreenControl system allows access to parameters from outside

```
class example : public sc_module {
 public:
  SC_CTOR( example ) :
    scparam ("scparam",  0xdeadbeef)
    {
      // ...
    }
  //...
private:
    gs_param<uint32_t> scparam;
}
```

Figure 4.1: Example use of gs_param<>

8 Initial work was done with Simics. Lately all work was enhanced and ported to QEMU.

the module that declares them, and even when the simulation is not running. *Figure 4.1* shows a short example of the use of parameters. Essentially, this provides the back-door access we need in order to get and set the simulation state.

When a gs_param<> is declared, it is given a parameter name, which is used to access it along with an automatic hierarchical naming scheme that mirrors the structure of the SystemC simulation set-up.

The main limitation to the above proposed system is that only SystemC module data members using the gs_param<> mechanism are checkpointed. In particular, signals and ports are not currently covered by our implementation. This means that signals and port values are not saved, and SystemC modules have to be written with this in mind. For modules written in a TLM-2 style, this is not a big issue since communication between modules is based on function calls that finish each transaction as a unit and that naturally store the updated module state in parameters marked with gs_param<>.

Another limitation is that we currently assume the SystemC set-up to be a fixed subsystem in the Virtual Platform system. The entire SystemC subsystem is represented as a single VP simulation model, with attributes reflecting the state of the SystemC kernel as well as the SystemC device models and the SystemC bridge itself. Since the names of parameters come from the SystemC hierarchy, this is also necessary to make parameter names meaningful.

However, note that this approach also means that the SystemC model implementation is separated out, and that different models can thus implement the same set of parameters. This supports such use-cases as changing the level of abstraction in a simulation and updating a model while reusing the same checkpoint data.

9 Simics and QEMU did support natively Checkpointing (when not supporting SystemC devices).

We call this strategy of managing data an implicit checkpointable model. On the other hand, we also propose and support an explicit checkpointable model, where we provide two new virtual functions for modules in SystemC (virtual functions to sc_module class[10]).

In the explicit checkpoint modules, the designer may implement two functions named **do_checkpoint()** and **do_restore()**. These two functions will be invoked by the checkpoint manager when starting a checkpoint or when recovering a simulation from a checkpoint data respectively. The first function is in charge of gathering and sending to the checkpoint controller all data needed to checkpoint the module. Symmetrically, the other function receives previously stored data in the checkpoint file to put them again into the module.

In most cases, implementing these two functions is quite simple and allows the adding of checkpoint support to existing SystemC descriptions with only light modifications to the existing source code.

4.3.2 SystemC kernel

4.3.2.1 Threads and Methods

One of the essential elements in SystemC is the model of concurrency. This is accomplished using *processes*. These processes are very similar to VHDL processes. In SystemC the processes are blocks of sequential code defined inside class members. These processes are not explicitly called by the user. Instead, they are defined inside each module and they are called by the SystemC simulator. SystemC defines two different processes, threads and methods. There is a third type named SC_CTHREAD but for all purposes in this work it is equivalent to SC_THREAD.

10 These two functions have the same nature like *start_of_simulation()*, *end_of_simulation()*…

METHODS

This type of process is completely executed from start to end every time the simulator invokes it without keeping any notion of internal state to the process. The user has two different options for the synchronization of the process: make it explicit in the process declaration, indicating to which *events* the process is sensitive (static sensibility) or using *next_trigger()* sentence inside the code (dynamic sensitivity).

The simulator will run again from the beginning of the *method* when event condition in the *next_trigger()* sentence is accomplished or events in the sensibility list are triggered.

THREADS

These kind of processes are executed by the simulator only once at the beginning of the simulation. Usually they are used with an infinite loop inside them in which the model can use the scheduler *wait()* method to synchronize *events* or to let some amount of time pass. The simulator will resume the execution of a suspended *thread* when event conditions are accomplished or the simulation time advanced to the correct time. The execution is resumed just after the *wait()* sentence in the *thread*.

4.3.2.2 Checkpointing Methods and Threads

SystemC is an event-based simulator that maintains the simulation state storing pending events for SC_METHODs and SC_THREADs in separate priority lists. When simulation begins, the kernel finds the top event of each list and executes or resumes the process sensible according to that event.

In case of SC_METHODs the sensitive process, which is essentially a method in the device model, is executed. Once the function call returns and the execution of the sensitive process ends, the SC_METHOD module is suspended until another event that it is sensitive to occurs.

A more complicated mechanism is used when managing SC_THREADs because a process can be suspended in the middle of its execution (when *wait()* is called). This requires the use of a user-level threading system to store the execution state of the model, including local variables on the stack. Execution is later resumed at the point where it was suspended, when an appropriate wake-up event happens.

The strategy we have followed taken in this work is to save and restore all information that the kernel needs to continue a simulation. For this, we need to save the actual simulation time and the event queues, primitive channels data, etc. This option will allow us to work with SC_METHODs only, because we cannot store and resume execution of SC_THREADs in middle of their execution.

The problem with SC_THREAD is not so much the SystemC kernel itself, as the fact that the kernel implementation uses a threading library that maintains a separate stack for each thread. It is infeasible to access, not to mention restore, the stack-based state of an SC_THREAD.

Fundamentally, the use of suspendable threads is inappropriate for checkpoint and restore. It puts state in stack-allocated local variables, as an implicit part of the call stack, and into processor registers. This means that state is not explicit and not available for manipulation from the outside.

It is known that use of SC_THREADs for Virtual Platform devices modeling is not recommended. Use of SC_METHOD instead of SC_THREAD to reduce the number of events and thus improve simulation speed is recommended in literature [50; 51].

We exploit these recommendations and we restrict our checkpointable SystemC to work with SC_METHODs only. This restriction shouldn't be a major problem in the Virtual Platform field, because several SystemC devices currently written may follow those guidelines to use SC_METHODs only and an SC_THREAD based device can be easily ported to use SC_METHODs.

The resume operation consists of constructing the SystemC subsystem again (as it would be starting a new simulation), and once the elaboration and initialization phases are done, we update the kernel state to the state saved in the checkpoint. In particular, the simulation time is changed, the pending events are reposted to the event queues, primitive channels are restored back with correct data, etc. Thus, the SystemC kernel will run the simulation as if it were just continuing the checkpointed simulation without any interruption.

4.3.2.3 Checkpointing Events

As the SystemC simulator is an event-based kernel simulator, we need to retrieve and save all event lists in the kernel and later restore them again. Once the event list is recovered and current simulation time is updated, the simulation kernel will act as if checkpoint and restore operations never happened, resuming the simulation normally.

We access the event queue through the OSCI SystemC kernel class *sc_simcontext*. This class encapsulates the SystemC kernel and is in charge of managing the simulation (basically managing event queues, advance simulation time and trigger process) and provides us with the simulation time and a set of handles to the event queues.

The list of events is communicated to the Virtual Platform as a module attribute of the SystemC bridge module. In order to access the list of active events and methods, we had to add some non-destructive inspection code to the OSCI SystemC 2.2.0 kernel.

In the case of the SystemC kernel, *sc_simcontext* manages four different event queues, two for events involving methods, and two for threads process. As we don't support checkpointing for threads, these last two queues are not taken in account. The other two queues are storing the pending event list for dynamic sensitivity and static sensitivity of the method process.

We add a single method to this queue mechanism in order to get all pending events without destroying the list: we had to add the method *get_elem(i)* to the SystemC utility classes *sc_ppq_base* and *sc_ppq* to non-destructively read out the list of pending events. In this way, once the user has checkpointed the simulation, he can continue the simulation as if nothing had happened.

When a checkpoint is taken in the Virtual Platform, the events queue lists are read and passed to the Virtual Platform checkpointing mechanism in the form of a formatted string, and simulation can continue normally.

When resuming from a checkpoint, these queue lists are posted back again, taking data from the previously saved set by the Virtual Platform.

The existing *sc_ppq::insert()* method was used to insert the saved list of events into the event queue when restoring operation.

Once we finish posting all events, the simulation state becomes as it would in the previous checkpointed data.

4.3.2.4 Checkpointing Primitive Channels

SystemC offers channels as a mechanism to communicate different modules or processes in a simulation. Similar to wires or signals found in other HDL, SystemC enables the separation of the module implementation from the communication scheme used in between modules as an essential part of system level modeling.

The SystemC library is given with a set of implemented primitive channels: sc_buffer, sc_fifo, sc_mutex, sc_semaphore, sc_signal, sc_signal_resolved, sc_signal_rv.

It's clear that these channels may be checkpointed with the rest of the system. So we need to add some back-door methods to enable checkpointing for each of them.

This work is done for the majority of the fabric primitive channels. Below is a detailed explnation of case for the sc_fifo<T>[11] channel.

sc_fifo<T>

sc_fifo is a SystemC predefined primitive channel modeling the behaviour of a FIFO (first-in, first-out) buffer. Each sc_fifo object has a number of slots to store values, and this number is fixed when the object is instantiated. sc_fifo<T> class is designed as a C++ template, meaning that it can manage any data-type using the same methods.

To allow checkpointing to sc_fifo<T> class, we added two new methods to the class: **do_restore()** and **do_checkpoint()**:

- std::string sc_fifo<T>::do_checkpoint() returns a string with all relevant data for the sc_fifo: number of slots and data stored in it.
- bool sc_fifo<T>::do_restore(std::string chkp_data) receives a string with all checkpoint data and restores back elements into the sc_fifo buffer.

11 <T> symbol denotes use of a C++ template, meaning that any data-type is allowed for that channel.

These two methods will work if data-type T supports converting to a meaningful string. If the user needs to use a special data-type without that feature, he can override these methods to implement its own special checkpointing methods[12].

It's important to emphasize that these new checkpointing methods don't disturb normal simulation of the system, nor penalize systems in simulation speed or any other aspect, because they are used only in the checkpoint or restore phases of the simulation.

Other channels

The same strategy has been used for other primitive channels present in the OSCI kernel suitable to be used in TLM-2 modeling, allowing them to be saved or restored by the virtual platform.

In this work, we don't modify channels like signal or buffer because they should be avoided in TLM-2 modeling due to the fact that their behaviour should be implemented in some way using sockets: for example, an output port of a module of type signal (that could emulate a interrupt signal in a RTL model) should be changed to a simple socket (for instance signal_socket) to emulate its behaviour in TLM-2 rather than maintaining the signal for that part of the model.

In any case, if necessary, implementation of checkpointing for sc_signal or sc_buffer is straightforward using the same strategy used in sc_fifo of adding those explicit methods for checkpoint.

12 These two methods are declared *virtual* so either of them can be overridden.

4.3.2.5 Checkpoint manager class

In order to enable checkpointing in the SystemC kernel, we introduced a new class into it. This class, named **chkp_mng**, manages all operations needed to perform the restore or checkpoint correctly.

This class is designed as a singleton to ensure that only one instance is called per simulation. This way, some of the modules in the system (usually the bridge to the Virtual Platform) can control the simulation and the checkpointing mechanism using the **chkp_mng** class methods. These methods include:

- std::string **chk_mng::checkpoint**() triggers the checkpoint process at current simulation point. This method returns a string with all checkpointing data. This string can be managed by the Virtual Platform as the essential checkpointing data of the SystemC based sub-system, store it to disk, etc.
- bool **chk_mng::restore**(std::string) to start process of restoring all events and data back into simulation to reach a previously saved state. The method receives a string with all checkpoint data. This data is retrieved and managed by the Virtual Platform, being independent of the implementation of the Virtual Platform.

The management of the data is left to the Virtual Platform, as we only use strings to store all checkpoint data (event list and simulation time, module state variables value, primitive channels, etc.). In this manner, the data and code we wrote to manage checkpoint is independent of the Virtual Platform used. So, even single SystemC simulations can make use of the Checkpoint feature.

This class is also responsible for calling the (if it exists) do_restore() or do_checkpoint() functions in any module implementing them to allow explicit checkpointing. Pseudocode for the do_checkpoint() function is depicted in *Figure 4.2*.

We added this new class to the existing OSCI simulation kernel version 2.2.0. This way, any application linked to SystemC kernel can use these new characteristics without any need to add new libraries, object files, etc.

```
function do_checkpoint() {
      get current simulation time

      for each (event in event_list) {
            get event time
            get SC_METHODs' name sensible to event
      }

      for each (module in system) {
            if (exists) call do_checkpoint()
            else get all gs_param<>
      }

      for each (prim_channel on system) {
            call do_checkpoint()
      }
}
```

Figure 4.2: Pseudocode for do_checkpoint() function

4.3.3 FSM[13]

Finite State Machines (FSM) are usually described in SystemC using SC_THREADs. To describe the Finite-State Machine depicted in *Figure 4.3*, the SystemC code using one SC_THREAD shown in *Figure 4.4* in page 98.

As seen in *4.3.2.1*, we cannot checkpoint threads, so we need to change this process to a SC_METHOD in order to be checkpointed.

To obtain a equivalent SC_METHOD from a SC_THREAD, it is necessary to add a new global variable to the process to store the state of the FSM and modify the code as shown in *Figure 4.5*. This new variable will be a data member of the SC_MODULE class. If we use gs_param<> type for that state variable, it will be checkpointing automatically by the simulator kernel when using our checkpointing manager **chkp_mng** class.

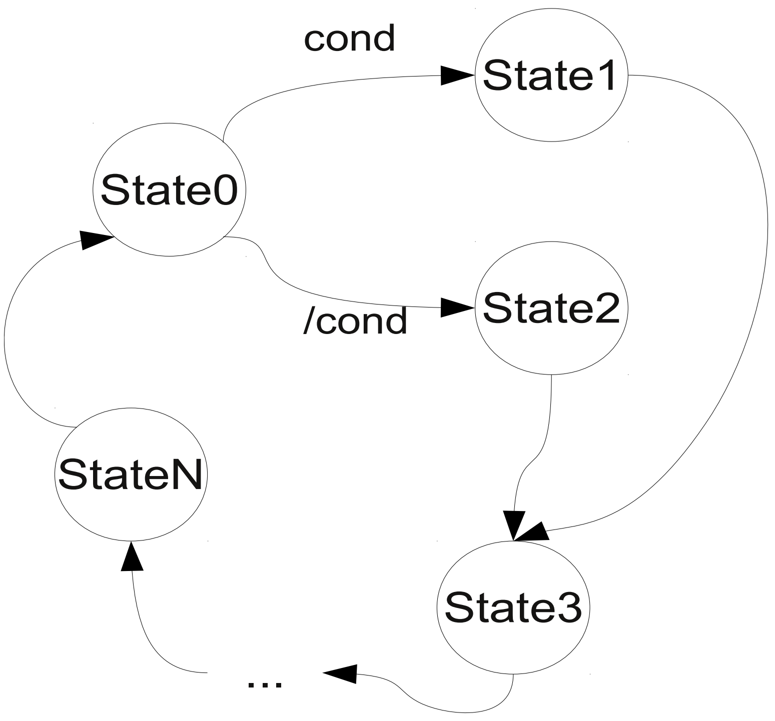


Figure 4.3: FSM example

13 Finite State Machine are not related to Flying Spaghetti Monster, but they use same acronym. Of course, FSM created FSMs long time ago.

```
SC_THREAD(my_fsm_thread);
sensitive << system_clock;

void my_fsm_method() {
      Sequential_Code_State0;
      wait()

      if (condition) {
            Sequential_Code_State1;
            wait()
      } else {
            Sequential_Code_State2;
            wait()
      }
      ...
      Sequential_Code_StateN;
      wait()
}
```

Figure 4.4: Code example for a SC_THREAD

```
SC_METHOD(my_fsm_method);
sensitive << system_clock;

void my_fsm_method(void) {
      switch (FSM_State) {
            case State0 :
            Sequential_Code_State0;
            if (condition) {FSM_State = State1;}
            else FSM_State = State2;
            break;
      case State1 :
            Sequential_Code_State1;
            FSM_State = State3;
            break;
      case State2 :
            Sequential_Code_State2;
            FSM_State = State3;
            break;
      ...

      case StateN :
            Sequential_Code_StateN;
            FSM_State = StateN;
            break;
      }

}
```

Figure 4.5: Code example for a FSM using SC_METHOD

Also, we shall change the code schema from a never-ending loop containing a set of instructions for a state, next state instructions, etc., (wait calls) to a new schema with a switch-case where every case of the switch contains the instructions for one of the states of the FSM.

With these modifications, every time that the SC_METHOD is executed, it will evaluate what is the state for the FSM and it will execute only that part of the code acting as the original SC_THREAD.

This trivial adaptation allows to re-write all SC_THREADs in a system description to its equivalent SC_METHOD process.

If some of the states of the FSM have a call to other function in the code, it may be managed in special way, ensuring that this other function do not call *wait()*.

4.4 Results

We used a set of different tests to validate our design and implementation of checkpointing for SystemC.

We work first on Simics SystemC bridge to add some preliminary checkpointing, as explained in next section.

Later, we used that work to add checkpoint to QEMU-SC and QBox. These tests and their results are explained in section *3.3.3*.

4.4.1 SC_THREAD to SC_METHOD overhead

We evaluated the simulation speed for different FSMs described using a usual SC_THREAD and translating later to a SC_METHOD to be used in our checkpointable SystemC. This experiment would be useful to observe if changing the FSM code from THREADs to METHODs would penalty worsen the simulation performance.

The translation was done by hand, firstly checking the correctness of the translation and later measuring simulation time for the equivalent description. In these tests we also included the description of the same FSM using gs_param<> to store the state for a implicit checkpointable description.

For each test, we wrote a test-bench for 200.000.000 transitions for the FSM and we used UNIX command *time* to get the total time spent in the simulation. This measurement was done three times per test and the results presented here are the arithmetic mean of these three measurements. We depict all results in *Table 4.1*.

Test #1 is a simple 3-state FSM with only 1-bit input signal and 1-bit output signal. This test should show the impact of the transformation of a SC_THREAD to a SC_METHOD. Also, it shows impact of a single variable stored in a gs_param<> when used to store the state variable of a simple FSM.

Table 4.1: Time results for FSM description using SC_THREAD and SC_METHOD

FSM	SC_THREAD	SC_METHOD	SC_METHOD with gs_param
Test #1	115,05	85,86	109,68
Test #2	130,68	119,64	195,22
Test #3	-	86,04	318,57

Test #2 is a multiplication by iteration implemented as a state machine. This module receives two 32 bits unsigned integer and performs the multiply between them adding and accumulating one factor by N times (where N is the other input factor). This test should show the impact of using some variables stored in gs_param<>.

Test #3 is a Fast Discrete Cosine Transformation module designed for JPEG compression [36]. This module is designed for Virtual Platform use, so it is designed to receive data to be processed in a single transaction; then the module performs the calculations and sends the result back to the VP. For this, this module is not using any SC_THREAD or SC_METHOD. We included it as a example of pure VP designed module and to show the overhead of using 5 intensive variables in gs_param<> although in a final implementation no state variables may be used.

As seen in *Table 4.1*, a FSM described using only a SC_METHOD, is about a 25% faster in simulation speed. On the other hand, using SC_METHOD and storing the state variable in a gs_param<>, reduces the performance to 95% of the original description.

To summarize this aspect of the results, it is clear that it is better to use explicit checkpointing[14] for the FSM itself instead of use *gs_param<>* if simulation time is very crucial, due to the fact that the translation from a SC_THREADed FSM to a SC_METHOD may be done by hand.

4.4.2 SystemC checkpoint support overhead

In our SystemC implementation of checkpointing, there is a potential additional performance impact coming from the use of gs_param<>, due to the GreenControl mechanisms.

14 Implementing the callback function **do_checkpoint()** in the FSM module.

We created a performance test [52] (cf. tbl. 1) consisting of two TLM-2 devices: (1) A master, that exercises 100 million TLM-2 write transactions to a (2) slave which writes the received payload to an unsigned int, a sc_uint or gs_param variables. This means an intensive use of parameters inside the tested module.

Table 4.2 shows that accessing checkpoint variables, which have been instrumented with gs_param<>, is quite efficient. The results show a penalty of about 2% (depending on the data type) as compared to the same model without internal variables modelled with gs_param<>.

As we seen in previous chapter, using gs_param<> for intensive use of variables (like state variable) can lightly penalize simulation performance.

For that reason, we added the explicit checkpoint mechanism to any module, giving the user the choice of either implementing the do_checkpoint() /do_restore() functions for any of the modules or using gs_param<> for implicit checkpointing.

It seems clear that using explicit checkpointing has no performance penalty, because data types and variables are unchanged, and only two function implementations are required.

Table 4.2: Performance test result for gs param

Data type	Runtime (s)	Penalty factor
unsigned int	15.48	1.00
gs_param<unsigned int>	15.76	1.02
sc_uint	15.79	1.02
gs_param<sc_uint>	16.18	1.05

4.4.3 Checkpoint size

Since our checkpoint system only stores the essential data for a simulated system, in general it will generate very compact checkpoints. For our running example, the checkpoint was about 88 kB in size – most of which is the contents of the RAM of the simulated machine containing code. The aspect of the information stored in a checkpoint for our test example is shown in *Figure 4.6*.

That can be compared to the overall process size of the simulation, at 263688 kB, which also includes overhead such as the simulation core, simulation code, and user interface system. Using the "store memory contents to disk" approach to checkpointing gets you to that size.

If the simulation system is placed inside a VMware virtual machine, and VMware snapshotting used to save the state, the size of that snapshot is the size of virtual RAM, which is at least 1GB for any reasonable simulation-hosting set-up.

```
OBJECT tgc0 TYPE timer_greencheckpoint {
      queue: cpu0
      build_id: 0x9e2
      systemc_time: 0x5c10c637307
      max_time_skew: 0x174876e800
      irq_dev: (pic, "internal_interrupts")
      irq_level: 23
      irq_pending: 0
      gs_all_param_value:
      "timer_greencheckpoint.interrupt_status=0;
      timer_greencheckpoint.register_bell=1;
      timer_greencheckpoint.register_control=1;
      timer_greencheckpoint.register_countdown=100000;
      timer_greencheckpoint.register_status=0;"
      methods_events_list:"6326694671111\n
      6426693333333;timer_greencheckpoint.timer_manager;"
}
```

Figure 4.6: Checkpoint information for the SystemC timer device in Simics

Other checkpoint implementation based on *process checkpointing* [29] presents results for a simple SoC[15] system using more than 400 MB for checkpoint data, growing linearly with the complexity of the design.

We should note here that size of the checkpoint is not a problem by itself (when it is limited to Megabytes or hundreds of Megabytes), but the problem appears when saving that data to disks.

Typical time values for a checkpoint in [29] are of seconds for a simple SoC, but the time is linear until file size breaks some O.S. parameter.

The checkpoint time result is key to the usability of the platform, mainly when there is a need to use *periodic checkpointing*. This feature allows designers to go back in simulation time restoring previous checkpointed points. If this restore and save times are too big, this feature is useless.

15 Example system is a basic PDA device.

4.4.4 Validating Checkpointing

To validate that we can indeed save and restore a Simics simulation including a SystemC subsystem, we used a fairly simple example device as shown in *Figure 4.7.* This device exhibits all essential problems for checkpointing, namely state in memory-mapped registers, and an SC_METHOD sensitive to an event that is posted at some point in the future using sc_notify.

Our test devices consists of a single memory-mapped register and a function sensible to an event. When the register is written, a periodic event is triggered at each fixed time (1 µs).

Our test began with starting the simulation and doing a write to the register to start the periodic event. Then, do a checkpoint and quit Simics. Then, start Simics again and resume the simulation from the checkpoint. We could observe that the periodic event is triggering again as expected. With this test, we validated that the SystemC kernel event list for SC_METHODs is properly saved and restored.

```
class systemc_greencheckpoint_test:
  public sc_module,
...
  SC_METHOD(function);
  sensitive << my_event;
...

void systemc_greencheckpoint_test::function()
{
      cout << "(SC code) function called at " <<
        sc_time_stamp() << endl;
      my_event.notify(1, SC_US);
}
```

Figure 4.7: Code for test device

4.4.5 Validating Model Updates

Model updates make it possible to change the code for a specific part of the model and re-use the same checkpoint data. A typical example (and what we use in our tests) is a device that needs one more register in its register file. Model update allows designer to take a checkpoint of the simulation, change the device register file by adding the missing register, and restore the simulation. In the continuation of the simulation, the model will recover the values for all registers previously present in the device.

To validate the updatability of a model with new features using a checkpoint for an older version, we created a simple SystemC device model containing a single memory-mapped register. Later we wrote a driver program for this device, executed the program to change the value of the register, and took a checkpoint. At this point, we exited the simulation, and changed the source code of the model to include an extra register. After recompiling, we started the simulation from the checkpoint without problems. The new register took on its default value as set in the SystemC source code, while the old register used the value provided in the checkpoint. We then executed the simulation for some more time, and took a new checkpoint, this new checkpoint correctly contained the state of both registers, as affected by the software driver program (the driver knew about both registers from the beginning). Thus, we show that we can update models and use old checkpoints while those models are compatible.

Te other side of updatability is the feature that allows designers to change some of the data stored in the checkpoint information to modify some of the variables checkpointed. In this way, a designer can change the value of a register of a device prior to restoring the simulation, change the value of a interrupt signal, etc. This feature saves lot of simulation time when multiple choices have to be tested, or when a bug is found that can easily be bypassed changing a single value (or a set of values).

As our checkpoint implementation manages checkpointing data in plain text (ASCII data), it is trivial for any designer or development tool to find and change values for a concrete variable of a device.

We also like to note that the model updatability is only possible with the kind of checkpoint we implemented; when using full process save for checkpointing, it becomes impossible to modify any of the parameters or variables of the simulation.

4.4.6 Completeness of checkpointable SystemC subset

Our implementation of checkpointing for SystemC must be useful for designers using this language to describe devices for Virtual Platforms. So we need to prove in some way, in spite of the fact that we only can checkpoint a sub-set of SystemC, that this sub-set is enough to describe any kind of device in the environment of Virtual Platforms.

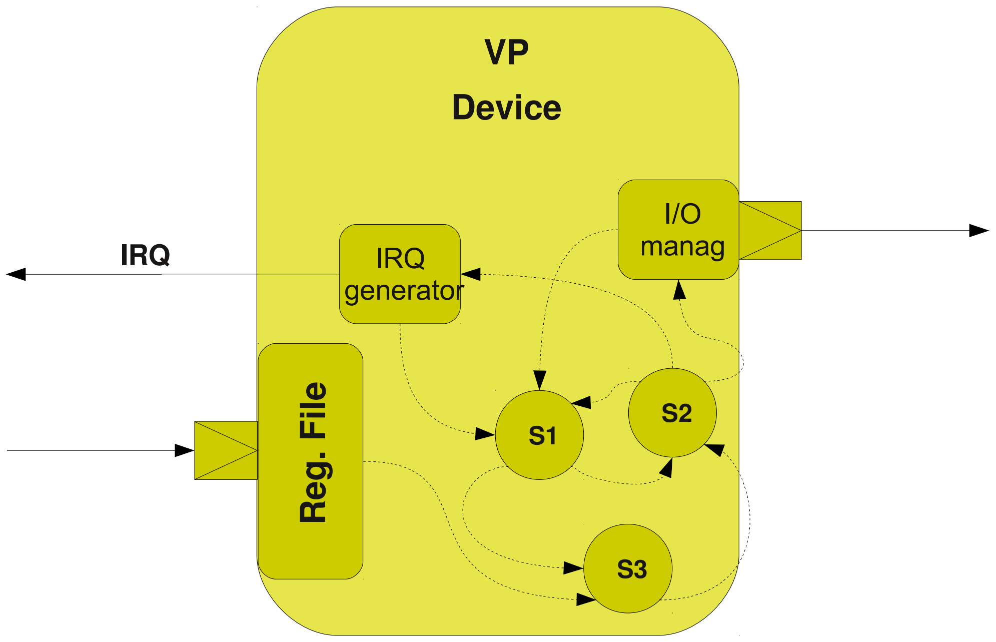


Figure 4.8: Typical VP device

Firstly, we must consider what type of device Virtual Platform usually use. Virtual Platforms usually emulate CPU-based systems (single or multi core), and these systems use memory mapped buses to connect devices to CPU. These devices are usually formed by a register file mapped to the bus and other circuitry performing the functionality of the device. Also, usually, accessing to the register file triggers some of the features of the device, while other functionality is usually triggered by interfacing outside the device. The basic structure of a typical Virtual Platform device is shown in *Figure 4.8*.

For the first part of these kind of devices (the register file), our proposed SystemC subset it is enough to describe the register file and the triggering of associated events to its access.

For the functionality side, as our SystemC subset is complete enough to describe FSMs, manage signals and their reactivity, it should be enough to describe any component of a Virtual Platform.

To illustrate this completeness, we wrote a checkpointable model of a PCI Ethernet card, composed of one register file and a set sub-modules to manage the communication and the construction of Ethernet packets to/from the TLM-2 sockets. This example was introduced and discussed in chapter *3.3.3* and depicted in *Figure 3.10* on page 79.

Other authors recommends avoiding the use of SC_THREADs to boost simulation speed performance [50; 51]. Their thesis is to try to minimize the number of events in the simulation changing the modeling style from SC_THREADs to SC_METHODs and they give some advice on how to achieve that. However, these authors don't prove that the use of only SC_METHODs is sufficient to model any possible device.

4.4.7 Simulation speed of the checkpointable OSCI kernel

To prove that the simulation speed of our modified OSCI kernel is not affected by the checkpoint mechanism, we simulated different systems with a fabric comparing unmodified OSCI simulator and our checkpointable kernel to measure simulation speed.

We use the same test systems presented in *4.4.1*. Simulation time results were the same for both simulators in all cases, concluding that our implementation of checkpointing doesn't affect simulation speed of the kernel.

4.5 Limitations and Discussion

Our work on checkpointable SystemC has proved to be a key-enabler for SystemC language to be used in Virtual Platforms.

The limitation to support only SC_METHODs, can be seen as a restriction to the designers to push them to write better and faster models like other authors recommend, keeping in mind that the aim of this checkpointable work (and its limitations) is oriented to the use of SystemC in Virtual Platforms, not for general purpose descriptions.

The job done transforms the SystemC OSCI simulator to be checkpointable with reduced changes in the code and without penalty in its simulation speed.

We propose two different approaches to allow SystemC devices be checkpointable: one which we named implicit checkpointing, where the designer uses special data-types to save state variables (those to be saved and restored); and another one, explicit checkpointing, where designer may write two helper functions to perform save and restore operations.

This second option seems to be better, and separates data management and checkpointing control from the model code. In any way, we left designer freedom to choose which of the two methods they prefer to use in any situation.

Finally, we show in a complex example how it is possible to use only SC_METHOD in a description, and how this example behaves correctly in all simulations we did, including those that we checkpointed in some point of the simulation time.

5 Discussion and Conclusions

5.1 Discussion

This work has proved that it is possible to do checkpointing in SystemC upon existing Virtual Platform frameworks, with a moderate effort. It has also exposed some issues in the SystemC design.

The key problem is the provision of a UNIX-style threaded execution model in SystemC, rather than an event-driven run-to-completion model. The threaded style encourages storing essential simulation state on the stack and as the location in the code, which is very hard to explicitly save and restore in a portable manner. Thank to the existence of SC_METHOD, a round style can be implemented in the current SystemC framework.

SystemC would have to be refined to make the model state as a first-class aspect of the language, and not just something implemented in arbitrary ways using C++ mechanisms as it is today. The connections between modules and the modules themselves present in a simulation would also have to be first-class items. Finally, the kernel would need a host-independent representation of the state. Especially if state has to be exchanged between different SystemC kernels from different vendors, such standardization is needed.

An alternative model to enable checkpointing is to define a SystemC Virtual Machine that compiles models to byte codes rather than to native code, thus providing a layer of indirection that can be used to dump and restore the system state without changing to models. Such work has been done for Java, for example.

Maybe a better option for SystemC could be became a very similar description for other programming language that use virtual machines to run, like Java, C# or similar.

5.2 Conclusions

We have presented our work on adding SystemC capabilities to the QEMU platform emulator and Virtutech Simics tool, along with the adding of a new feature to the OSCI SystemC kernel to allow better integration with current Virtual Platforms.

We use TLM-2 for communication between QEMU and SystemC devices. This work allows designers to plug devices written in SystemC into some of the platforms that QEMU emulates.

The main problem that appears when joining two different simulators (OSCI SystemC simulator and QEMU) is time synchronization, and we detailed how we handled this in two different solutions:

- making the OSCI simulator a slave of the QEMU emulator, and running SystemC simulation only when it is really needed.
- Wrapping Virtual Platform to behave as a standard TLM-2 Initiator. This wrapper allows designers to use standard SystemC based virtual platform tools to model the entire system, with no special requirements.

We also addressed the provision of DMI support in our QEMU to SystemC bridge for devices that support it.

We have discussed the need and utility of checkpointing. It is an enabler for software development, and if it can be achieved in such a way as to allow the result to be made use of on different host platforms, it can be used in a number of ways.

In the past, checkpointing has been achieved by saving the complete process state of the process running the simulation. This limits the ability to distribute the simulation, and it is also expensive in terms of disk and time.

Our aim has been to investigate the possibility of saving and restoring the SystemC kernel and model state itself.

In order to achieve this ambitious aim, we had to limit ourselves in terms of what models we support. However, we have found, even with those limitations, the results are helpful and allow complete system descriptions.

We are able to save and restore the state of SystemC methods, and the events that trigger them. We are also able to save and restore the state of models that are running on the SystemC simulator.

This task has been achieved by introducing a new class to SystemC source code (the checkpoint manager) responsible for managing all necessary data to perform the checkpoint process.

This checkpointing strategy allows tools and designers to use model update in a checkpoint, that is a important feature that other important tools cannot currently offer because of the checkpoint mechanism they are using.

In the future we will be investigating other approaches to checkpointing, and dealing with the case of SystemC threads (for example, it could be possible to converting threads to methods using automatic tools [58]), although SC_METHODs-only modeling is enough for Virtual Platforms frameworks.

5.3 Impact of the QEMU & SystemC

When the QEMU-SystemC research project started, few or none of the emerging Virtual Platforms tools had support to SystemC or TLM-2.

In early 2007 TLM-2 just appeared with the intention to standardizing the Transaction Level Modeling in SystemC, keeping in mind the possibility to use it in actual or future Virtual Platform tools.

When we freely published our QEMU-SC work in the web, we received lot of feedback from people searching a similar solution, and happy to use our implementation. These people come mainly from universities, some wanting to use it for lab practices, others to use it as platform for their research. This feedback took the form of bug reports, patches, questions, suggestions, etc.

A time after that, around 2008, GreenSocs ask me to do a very similar task for Virtutech Simics tool. We decided to did it in parallel to implementation for QEMU, freely available and open-sourced. We added checkpointing to this bridge as described in previous chapters. From the Simics side, the job became the Simics SystemC bridge that then Virtutech (now Wind River) is offering with Simics framework until now.

The QEMU task was developed with the same quality and features requirements as Simics, and later we included the checkpointing mechanism used in Simics.

Then we changed our strategy and we wrapped QEMU to be a TLM-2 Initiator, founding the QBox project. This project was also used by Synopsys to test and show a demo of the just bought Innovator tool by CoWare in DAC'09 conference [44].

Actually, some research projects at various Universities and Research Institutions use or base part of their work on our QEMU-SC. These other authors published some conference papers [53-55] and one Master of Science in Engineering [56].

An indian company with focus on System on Chip (SoC) modeling and Embedded software services using Virtual Platforms includes QBox expertise in its portfolio [57]. This company is a GreenSocs partner, and was involved in the development of the QBox project.

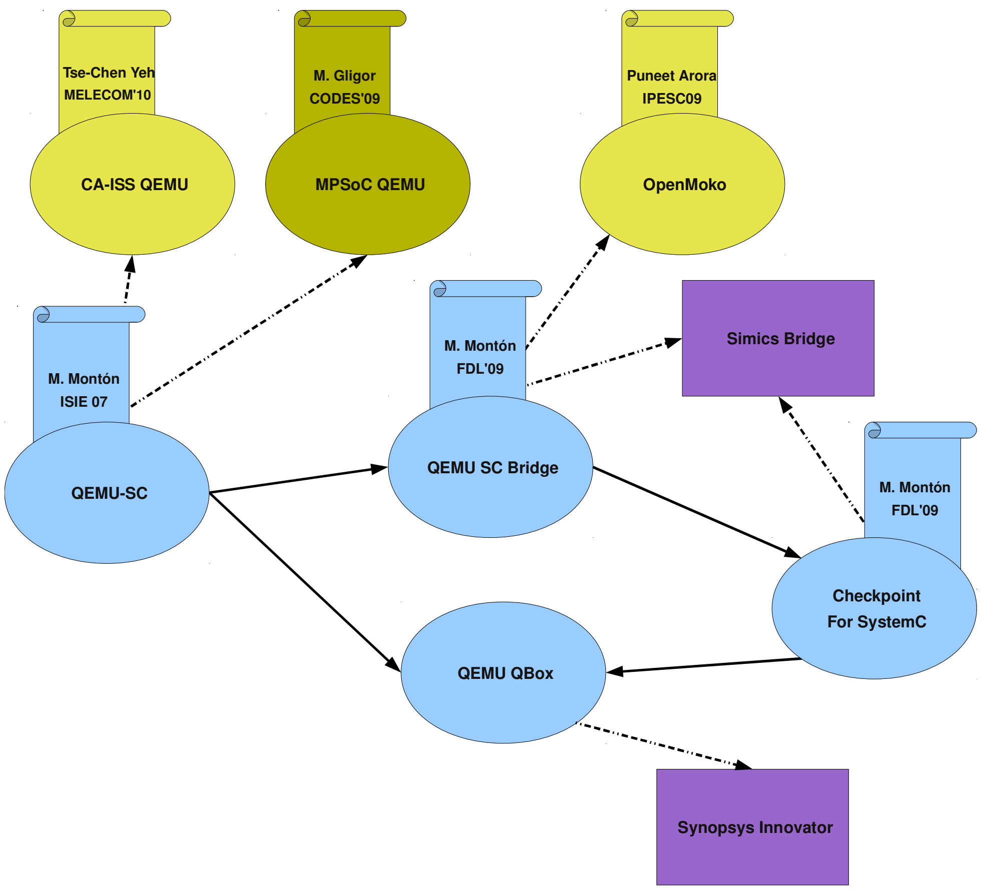


Figure 5.1: QEMU Ecosystem. In dotted arrows relation between our projects and third-party research projects (in yellow) and industry projects (in magenta).

The author has been invited to give a talk in the 1st International QEMU Users Forum in conjunction with DATE 2011 on March 18th, 2011. The presentation will be entitled “QEMU and SystemC”.

In *Figure 5.1* we summarize different projects and publications related to our work on QEMU, SystemC and Checkpointing, as well as the commercial tools and the relationship between them.

Curriculum Vitae

Màrius Montón was born on July 28th, 1976 in Barcelona, Catalonia. He received his Engineering degree in Computer Science in 2003 from the Universitat Autònoma de Barcelona (UAB) and he received a M.S. degree in microelectronics in 2006 from the same university. In 2000 he joined CEPHIS technology transfer node from the regional IT network as a R&D engineer. Since 2004 he is part-time teacher at the Departament de Microelectrònica i Sistemes Electrònics in the UAB.

At 2007 he joined GreenSocs, working on joining SystemC and TLM-2 with other simulators, developing projects for Virtutech (actually Wind River) and Intel among other collaborations including Synopsys and CoWare and other major EDA vendors.

He can be contacted at marius.monton@gmail.com.

Publications

During my university life, I have been involved in several industrial projects, research projects and collaborations with other people that lead to publications. There follows a tiny selection of most important papers published during those years.

Also, teaching experiences with other colleagues were resumed in some teaching papers.

Research Publications

Journals

1. Màrius Monton, Jakob Engblom, Mark Burton, "Checkpointing for Virtual Platforms and SystemC-TLM", In *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*. SUBMITED UNDER REVISION.

Book chapters

1. M. Montón, J. Engblom, C. Schröder, J. Carrabina and M. Burton, "Checkpoint and Restore for Systemc Models", In Bornione D., editor. *Advances in Design Methods from Modeling Languages for Embedded Systems and SoC's*, Vol 63. Springer 2010; (Lecture Notes on Electrical Engineering; vol 63). ISBN: 978-9048193035.
2. Casanovas, P.; Binefa, X.; Gracia, C.; Montón, M.; Carrabina, J.; Serrano, J.; Blázquez, M.; López-Cobo, J.M.; Teodoro, E.; Galera, N.; Poblet, M., "The e-Sentencias prototype. Developing ontologies for legal multimedia applications in the Spanish Civil Courts". In Casanovas, P.; Breuker, J.; Klein, M.; Francesconi, E. (eds) *Channeling the legal informational flood. Legal Ontologies and the Semantic Web*. IOS Press, 2009, 188, 199-219.

Conference papers

1. M. Montón, J. Engblom, M. Burton, "Checkpoint and Restore for SystemC Models" in *Proceedings of International Forum on Specification and Design Languages (FDL)*, Sophia Antipolis, France, Sep. 2009, pp. 1-6.
2. M. Montón, J. Carrabina, M. Burton, "Mixed Simulation Kernels for High Performance Virtual Platforms", in *Proceedings of International Forum on Specification and Design Languages (FDL)*, Sophia Antipolis, France, Sep. 2009, pp. 1-6.
3. M. Montón, B. Martínez, J. Carrabina, "Síntesis de Canales TLM para procesador Nios-II" in *VIII Jornadas de Computación Reconfigurable y Aplicaciones, (JCRA),* Madrid, Sep., 2008.
4. B. Martínez, J. López, M. Montón, J. Carrabina, "Sistema de Control Acceso por Reconocimiento Óptico de Documentos sobre Nios-II", in *VIII Jornadas de Computación Reconfigurable y Aplicaciones, (JCRA),* Madrid, Sep., 2008.
5. P. Rujan, F. Vuillod, B. Gomm, M. Montón, D. Castells. "AMASS Core: Associative Memory Array for Semantic Search" in *IP Based Electronic System Conference & Exhibition*. Dec. 2007. Grenoble, France.
6. C. Gracia, P. Casanovas, J. Carrabina, X. Binefa, E. Teodoro, M. Montón, N. Casellas, C. Montero, N. Galera, J. Serrano, M. Poblet, "Legal Knowledge Acquisition and Multimedia Applications", in *Knowledge acquisition from multimedia content Workshop (KAMC'07). Second International Conference on Semantic and Digital Media Technologies, SAMT 2007,* Genova, Italy, Dec., 2007.
7. M. Montón, J. Carrabina, C. Montero, J. Serrano, X. Binefa, C. Gracia, M. Blázquez, J. Contreras, E. Teodoro, N. Casell, J.J. Vallbé, M. Poblet and P. Casanovas, "Accelerating Semantic Search with Application of Specific Platform" in *Workshop on Semantic Web Technology for Law (SW4Law),* Stanford University, USA, June, 2007.
8. X. Binefa, C. Gracia, M. Montón, J. Carrabina, C. Montero, J. Serrano, M. Blázquez, R. Benjamins, E. Teodoro, P. Casanova, M. Poblet, "Developing ontologies for legal multimedia applications", in *Legal Ontologies and Artificial Intelligence Technique (LOAIT Workshop),* Stanford University, USA, 2007.
9. M. Montón, A. Portero, M. Moreno, B. Martínez, J. Carrabina, "Mixed SW/SystemC SoC Emulation Framework" in *Proceedings of IEEE International Symposium on Industrial Electronics. (ISIE)*. Vigo, June 2007, pp. 2338-2341.
10. A. Portero, G. Talavera, M. Montón, B. Martinez, J. Carrabina, "NoC System for MPEG-4 SP using heterogeneous tiles", in *XXI Conference on Design of Circuits and Integrated Systems. (DCIS),* Barcelona 22-24 November 2006.
11. A. Portero, G. Talavera, M. Montón, B. Martinez, M. Moreno, F. Cathoor, J. Carrabina, "Energy-aware MPEG-4 Single Profile in HW-SW Multi-platform implementation", in *IEEE International SOC Conference,* Sep., 2006, pp 1316..

12. B. Martínez, M. Montón, J. Carrabina, "Síntesis de Unidades Funcionales para Soft-Cores desde un modelo C/C++", in *VI Jornadas de Computación Reconfigurable y Aplicaciones, (JCRA),* Càceres, Sep., 2006.
13. A. Portero, G. Talavera, M. Montón, B. Martinez, F. Cathoor, J. Carrabina, "Dynamic Voltage Scaling for Power Efficient MPEG4-SP Implementation", in *Proceedings of the IEEE 17th International Conference on Application-specific Systems, Architectures and Processors (ASAP),* Washington, DC, USA: IEEE Computer Society, 2006, pp. 257-260
14. L. Ribas-Xirgo, D. Castells, M. Monton, J. Carrabina, "A Rapid Sorting Unit based on Programmable Shifting Register Files", in *XX Conference on Design of Circuits and Integrated Systems. (DCIS),* Lisboa, Portugal, Nov., 2005.
15. M. Montón, O. Font, J. Joven, P. García, L. Terés, J. Carrabina, "XML specification and tools for Automatic SoC Generation", in *XIX Conference on Design of Circuits and Integrated Systems. (DCIS),* Bordeaux, France, Nov., 2004.
16. D. Castells, M. Montón, R. Pla, D. Novo, A. Portero, O. Navas, J. Farré, L. Ribas, J. Carrabina, "Comparing Design Flows for Structural System Level Specifications facing FPGA Platforms", in *XIX Conference on Design of Circuits and Integrated Systems. (DCIS),* Bordeaux, France, Nov., 2004. .
17. D. Castells-Rufas, M. Montón, L. Ribas, J. Carrabina, "High performance Parallel Llinear Sorter Core Design", in *GSPx, The international embedded Solutions Event,* Santa Clara, CA, Sep., 2004.
18. J. Carrabina, M. Montón, R. Martínez, J. Joven, O. Font, R. Ruíz, P. García, L. Terés, "Bus-centric SoC Architecture Generation Tools", in *GSPx, The international embedded Solutions Event,* Santa Clara, CA, Sep., 2004.

Teaching publications

1. X. Fitó, G. Talavera, B. Lorente, M. Montón, B. Martínez, C. Ferrer, E. Valderrama, "Cas pràctic d'adaptació metodològica a les directrius EEES d'una assignatura d'enginyeria Informàtica", in *III Jornada de Campus d'Innovació Docent,* UAB, Barcelona, Spain, 2006.
2. G. Talavera, X. Fitó, B. Lorente, A. Portero, M. Montón , B. Martínez, J. Oliver, C. Ferrer, L. Ribas, J. Aguiló, E. Valderrama, "Adaptación metodológica a las nuevas directrices del EEES en la enseñanza tècnica universitaria", in *Tecnologías Aplicadas a la Enseñanza de la Electrónica (TAEE),* Madrid, Spain, 2006.
3. G. Talavera, B. Lorente, M. Montón , B. Martínez, J. Oliver, C. Ferrer, L. Ribas, J. Aguiló, E. Valderrama, "Nuevas metodologías docentes y autoaprendizaje en la enseñanza técnica universitaria", in *IV Congrès Internacional de Docència Universitària i Innovació. (CIDUI),* Barcelona, Spain, 2006.

References

1. IEEE Std 1666-2005: IEEE Standard SystemC® Language Reference Manual. 2005.
2. "Open SystemC Initiative". http://www.systemc.org.
3. Requirements specification for TLM 2.0, Version 1.1. 2007.
4. "Conventional PCI". http://www.pcisig.com/specifications/conventional/.
5. AMBA Specifications (Rev 2.0). Datasheet. 1999.
6. The CoreConnect Bus Architecture. 1999.
7. OSCI TLM-2.0 LANGUAGE REFERENCE MANUAL. 2007.
8. "ARM Versatile website". http://www.arm.com/products/DevTools/VersatileFamily.html.
9. M. Chung and C. Kyung. "Improvement of Compiled Instruction Set Simulator by Increasing Flexibility and Reducing Compile Time," In *IEEE International Workshop on Rapid System Prototyping*. 2004.
10. M. Reshadi, P. Mishra and N. Dutt. "Instruction set compiled simulation: a technique for fast and flexible instruction set simulation," In *Proceedings of Design Automation Conference (DAC'03).*. 2003.
11. F. Bellard. "QEMU, a fast and portable dynamic translator," In *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*. 2005.
12. P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg et al., "*Simics: A Full System Simulation Platform*," IEEE Computer **35 no. 2**, 50 (Feb 2002).
13. "Imperas webpage". http://www.ovpworld.org.
14. "CoWare webpage". http://www.coware.com/products/virtualplatform.php.
15. "Cadence System Design and Verification". http://www.cadence.com/products/sd/Pages/default.aspx.
16. "EDGE SimTest Webpage". http://www.mentor.com/products/embedded_software/edge-dev-suite/simtest/.
17. "Innovator Webpage". http://www.synopsys.com/tools/sld/virtualprototyping/pages/innovator.aspx.
18. "Bochs webpage". http://bochs.sourceforge.net/.
19. "VMWare webpage". http://vmware.com/products.
20. "Virtual PC webpage". http://www.microsoft.com/windows/products/winfamily/virtualpc.

21. "Kernel Based Virtual Machine". http://www.linux-kvm.org/page/Main_Page.

22. "Android Emulator". http://developer.android.com/guide/developing/tools/emulator.html.

23. Engblom J, Kagedal D, Moestedt A and Runeson J. "Developing embedded networked products using the Simics full-system simulator," In *Personal, Indoor and Mobile Radio Communications, 2005. PIMRC 2005. IEEE 16th International Symposium on*. 2005.

24. Imperas Software Ltd., "*OVPsim and Imperas CpuManager User Guide*," , (2010).

25. Imperas Software Ltd., "*Using OVP Models in SystemC TLM2.0 Platforms*," , (2010).

26. "Virtutech Announces Simics Full System Checkpointing for SystemC-based Transaction-Level Modeling". http://www.systemc.org/news/pr/view?item_key=8e49ae45a4f5f2506e33e62fe059ad298e47889d.

27. "OVP Simulator Smashes SystemC TLM-2.0 Performance Barrier". http://www.imperas.com/archives/157.

28. M. Bergqvist, J. Engblom, M. Patel and L. Lundegård. "Some Experience from the Development of a Simulator for a Telecom Cluster (CPPemu)," In *10 th IASTED Conference on Software Engineering and Applications* . 2006.

29. S. Kraemer, R. Leupers, D. Petras and T. Philipp. "A checkpoint/restore framework for systemc-based virtual platforms," In *International Symposium on System-on-Chip (SoC)*. 2009.

30. "SystemC Save and Restore Part 2 - Advanced Usage". http://www.cadence.com/community/blogs/sd/archive/2009/03/09/systemc-save-and-restore-part-2-advanced-usage.aspx.

31. M. Montón, A. Portero, M. Moreno, B. Martínez and J. Carrabina. "Mixed SW/SystemC SoC Emulation Framework," In *IEEE International Symposium on Industrial Electronics (ISIE07)*. 2007.

32. A. Portero, G. Talavera, M. Montón, B. Martínez and J. Carabina. "NoC System for MPEG-4 SP using heterogeneous tiles," In *XXI Conference on Design of Circuits and Integrated Systems. (DCIS'06)*. 2006.

33. A. Portero, G. Talavera, M. Monton, B. Martinez, M. Moreno et al.. "Energy-Aware MPEG-4 Single Profile in HW-SW Multi-Platform Implementation," In *Proc. IEEE Int. SOC Conf.* 2006.

34. "GreenSocket website". http://wnww.greensocs.com/en/Projects/GreenSocket/.

35. "MPEG Software Simulation Group". http://www.mpeg.org/MSSG/.

36. C. Loeffler, A. Ligtenberg and G. S. Moschytz. "Practical fast 1-D DCT algorithms with 11 multiplications," In *Acoustics, Speech, and Signal Processing, 1989. ICASSP-89., 1989 International Conference on*. 1989.

37. PC16550D Universal Asyncrhonous Receiver/Trasnmitter wifh FIFOs. 1995.

38. "Simics Virtual Platform for the AMCC Ebony (PPC440GP)". http://www.virtutech.com/solutions/virtual_platform/powerpc/amcc/ebony.html.

39. "Wind River to Add Virtutech Simics Products to Comprehensive Embedded Software Portfolio". http://www.windriver.com/news/press/pr.html?ID=7841.

40. "Simics SystemC Bridge". http://www.virtutech.com/systemc_bridge.html.

41. "DP8390D - NIC Network Interface Controller ". http://www.national.com/opf/DP/DP8390D.html#Overview.

42. "GreenRouter". http://www.greensocs.com/en/Projects/GreenParts/GreenRouter.

43. "Importing Wrapper for QEMU in Innovator". http://www.greensocs.com/en/Projects/QEMUSystemC/docs/ImportingWrapperforQEMUinInnovator.

44. "Synopsys at DAC 2009". http://www.synopsys.com/Archive/DAC2009/Pages/TheaterSchedule.aspx.

45. J. L. Peterson, P. J. Bohrer, L. Chen, E. N. Elnozahy, A. Gheith et al., "*Application of full-system simulation in exploratory system design and development*," IBM Journal of Research and Development, Vol 50, no 2/3 Issue 2/3, 321 (2006).

46. M. Rosenblum and M. Varadarajan, "*SimOS: A Fast Operating System Simulation Environment*," Stanford University technical report CSL-TR-94-631, (July 1994).

47. "Boost Serialization v.1.36". http://www.boost.org.

48. "CoWare Introduces First Ever Checkpoint/Restart Capability for Native SystemC Virtual Platforms". http://www.soccentral.com/results.asp?EntryID=25416.

49. "GreenControl website". http://www.greensocs.com/en/projects/GreenControl.

50. M. Schnieringer and K. Brand, "*SystemC: Key modeling concepts besides TLM to boost your simulation performance*," VaST Systems Technology, ().

51. S. Swan, "*An Introduction to System Level Modeling in SystemC 2.0*," Cadence Design Systems, Inc., ().

52. C. Schröder, W. Klingauf, R. Günzel, M. Burton and E. Roesler. Configuration and control of SystemC models using TLM middleware. , 2009.

53. M. Gligor, N. Fournel and F. P\'etrot. "Using binary translation in event driven simulation for fast and flexible MPSoC simulation," In *CODES+ISSS*. 2009.

54. T. Yeh, G. Tseng and M. Chiang. "A fast cycle-accurate instruction set simulator based on QEMU and SystemC for SoC development," In *MELECON 2010 - 2010 15th IEEE Mediterranean Electrotechnical Conference*. 2010.

55. P. Arora, R. Bharti and M. Burton, "*Creation of Virtual Platform Using OCP-IP Modeling Kit*," IP-ESC, (2009).

56. P. E. Salinas. Integration of Virtual Platform Models into a System-Level Design Framework. University of Texas at Austin. 2010.

57. "Circuitsutra Webpage". http://www.circuitsutra.com/.

58. "SCThreadConverter". http://www.greensocs.com/en/projects/SCThreadConverter.

Alphabetical Index

AMBA.....62, 64
Approximately-Timed.....**35,** 71
ARM.....41, 63pp.
AT.....60, 71
Blocking.....**33, 36**
Bochs.....40p.
Boost.....84
Cadence.....**40, 48, 51, 84**
Checkpoint.....51, 73
Chkp_mng.....95
Coldfire.....41
CoWare.....40p., 84
DMI.....63, 65
Dynamic sensitivity.....89
FSM.....**97, 99pp.**
Global quantum.....**61, 77**
GNU/Linux.....41, 63
GreenControl.....86, 101
GreenSocket.....62
GreenSocs.....62
Gs_param.....86p., 101
GSGPSocket.....62
Imperas.....**40, 48p.**
Innovator.....**40**
Intel.....41, 63
Loosely Timed.....**34,** 71
LT.....60, 71
Mambo.....**83**
Method.....88
MIPS.....41
MMU.....**45**
Non-blocking.....**33, 36**
Open Virtual Platform.....40
Operating System.....41
OSCI.....29p., 58p., 62, 64, 85, 91
PCI.....57, 62pp.
QBox.....75, 80p.
QEMU.....40pp., 55, 57pp., 65, 69p., 75p., 79
Quantum.....**61**
Quantum Keeper.....61, 77
Sc_fifo.....**93**
SC_METHOD.....64
SC_THREAD.....97
Sensibility list.....89

Simics...24, 40p., 45pp., 51, 69pp., 78, 85, 99, 105, 114, 116
SimOS...**83**
SimTest...**40, 48**
SPARC...41
State of the Art...29
Static sensibility...89
Synopsys...**40, 48, 80, 116**
SystemC...25p., 29p., 42, 51, 55pp., 69p., 73pp., 79, 83pp., 101, 105p.
Temporal Decoupling...**36**
THREAD...84, 88, 97
TLM...25, 30
TLM-2...30, 56p., 59pp., 64, 76, 83, 87
UART...62, 74
Untimed...**34,** 71
UT...71
Virtual PC...**40**
Virtual Platforms...48, 51
Virtutech...40, 69, 85
VMware...40, 85, 103
Wait()...89p.
Windows XP...41