

Mixed simulation kernels for high performance virtual platforms

Màrius Montón[§]

GreenSocs
http://www.greensocs.com

marius.monton@
greensocs.com

Jordi Carrabina

Dpt. Microelectrònica i
Sistemes Electrònics
Universitat Autònoma de
Barcelona

Jordi.carrabina@uab.cat

Mark Burton

GreenSocs
http://www.greensocs.com

mark.burton@
greensocs.com

Abstract

We present work in the domain of Virtual Platforms, based on the QEMU emulator. Virtual Platforms allow software and drivers to be developed in parallel with the development of hardware, avoiding re-design and long delay times in SW development. This work allows designers to plug SystemC models into the virtual platforms that QEMU offers (We focused on two of the available platforms: x86 PC and ARM's VersatilePB) The new aspect of this work is the technology we have developed to connect between QEMU and SystemC. We have developed a virtual device to link QEMU and SystemC, and a bridge to manage the OSCI SystemC-2.2.0 simulator. This bridge accomplish the task of synchronize efficiency the two simulators, using a strategy of freeze-and-update on the SystemC simulator to achieve a good performance. Connection with the SystemC device is done using TLM-2.0 sockets and makes use of DMI. Also we present the same emulator wrapped for a TLM-2.0 Initiator module. With this wrapper, this QEMU module can be used in a standard SystemC simulation environment as an Initiator that accesses some (but not necessary all) of its system devices through a standard TLM-2.0 socket.

1. Introduction

A virtual system platforms 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.

Several products can be included in this definition, e.g. Virtutech's simics [1], Imperas' Open Virtual Platform [2], CoWare Virtual Platform [3], QEMU [4]

Bochs [5], VMWare [6] and Microsoft Virtual PC [7]. Some of these (e.g. Simics, OVP and 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 normally they are not 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.

The work described in this paper adds more functionality to the open sourced QEMU emulator to help ESL design. We focused on adding support for SystemC devices to QEMU, allowing designers to easily test new devices in the final platform, including O.S., device drivers and applications.

Our previous work on this topic [8] was based on modifying QEMU so that a model that would later be synthesized, could be used within QEMU. We achieved this with an RTL connection between QEMU and the SystemC model, in order to help designers to develop correct SystemC models to be later synthesized using high level tools like Cynthesizer by ForteDS [9]. The solution as proposed focused on the connection – but was not very efficient. For many ESL Virtual Platforms, performance is very important.. The Current work is focused on giving a complete virtual platform using a high level TLM-2.0 description instead of the older RTL level connection.

2. 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. (lot of other platforms are in

[§] Also PhD student at the Universitat Autònoma de Barcelona, Barcelona. Spain.

the development stage). The emulated systems can run unmodified Operating Systems (guest O.S.) like GNU/Linux or MS-Windows XP on different hosts O.S.

QEMU consists of different 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 register 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.

3. QEMU-SystemC bridge

We developed a standard QEMU device (sc_link.c) that register itself as system bus device (AMBA or PCI). This module is in charge of capturing the bus access that it receives, and sending the data across the *SystemC bridge*. For a read operation, this link returns the data that is read from the SystemC device.

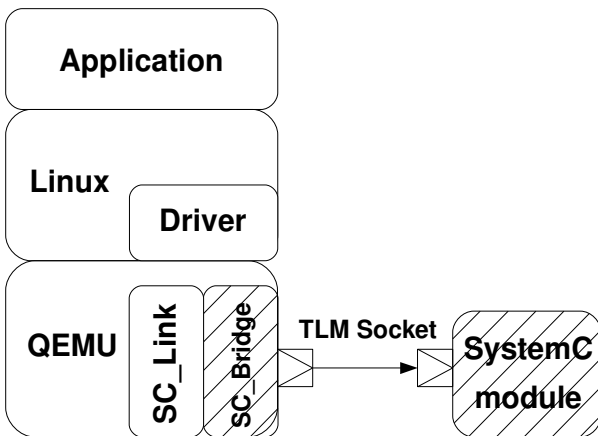


Figure 1: Block Diagram (SystemC blocks grayed)

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 Initiator for the transactions, and the SystemC device as the Target. Once called, the bridge creates and fills the TLM transaction and performs the transaction operation, managing SystemC simulation time to finish the transaction. The block diagram is shown in Figure 1, with SystemC blocks in gray.

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 it's specific bus fabric). The sc_link will do the conversion between that platforms bus access mechanism and the data necessary to call the *SystemC bridge* functions (basically, address and data) as depicted in Figure 2. The *SystemC bridge* module will actually construct and send the TLM-2.0 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 in advance what the address of our device is. This calculation is performed in the SystemC link, so the bridge always receive the operation data in the form of address and data.

This partition give 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.

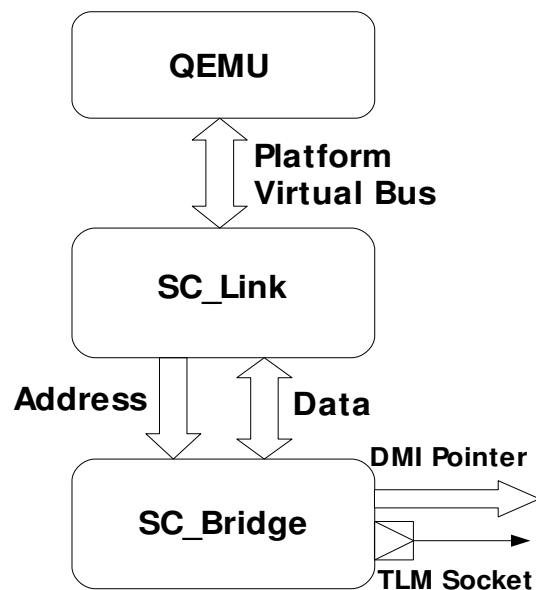


Figure 2: QEMU – SystemC Dataflow

3.1. Synchronization

The main problem when joining two simulators (like QEMU and SystemC) together is 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 solution is based on a lazy connection, and the notion of ‘quantum time’ introduced by OSCI’s TLM-2.0 standard. We synchronize both worlds only when it is necessary to do so. In the simplest case, we could imagine 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-2.0 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* ask the SystemC simulator for the next event in the simulation.

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 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 ‘quantums’ introduced by TLM-2.0, 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 event 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 have external I/O, because the device is “frozen” most of the time, the I/O operations may need to handled

by the *SystemC bridge* in order to gain enough responsiveness to the external input. This can be arranged notifying the *SystemC bridge* when a external event is happens, so that it can manage the SystemC simulator correctly. 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.

3.2. TLM socket

Communication between the bridge and the SystemC device is done through a TLM socket. We use the GreenSocs Generic Protocol Socket [10]. This socket allows connection of OSCI TLM-2.0 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 the possibility 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, enabling 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 noticeable 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 data necessary for each access to the device that is done. This involves filling the fields corresponding to access type (Read or Write), the address to access to, and the data to write in case of write access. The rest of the transaction is static throughout the simulation.

3.3. DMI

The TLM-2.0 Direct Memory Interface (DMI) is a specialized interface that provides direct access to a memory area within a target. This interface can accelerate memory accesses because it allows direct access to the memory, rather than having to pass through standard socket transactions.

If a bus is capable to 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. We implemented this 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).

4. Experimental 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

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 [11].

We started with a simple SystemC device that publishes a register file. This register file is accessed through the TML 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 sends the operation to the virtual platform system bus.

With this initial setup we can check that all components were running as expected, and that both simulators are properly well synchronized.

The next experiment was to add to the SystemC device the capability of generating an interrupt 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

```

...
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: IRQ generation code (a) constructor (b) SC_METHOD

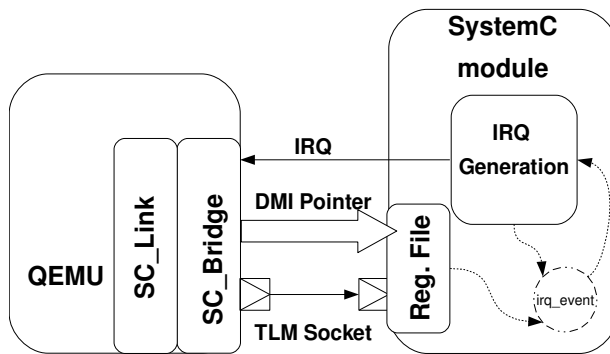


Figure 4: Final test bench module

sensitive to that event that triggers the interrupt and post the event again (Figure 3).

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

Our last 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 a memory devices than simply using standard transactions across the Sockets.

The results of this last experiments didn't show a performance enhancement of the DMI mechanism. We think this is because QEMU accesses to devices using single word accesses, instead using bus bursts. This issue is under study, and we will present some final results in the final version of this paper.

The resulting device and setup is shown in Figure 4, notice we develop the same tests for the ARM platform and for the x86 platform, having to modify only the SC_link module for each different platform and reusing all other modules.

5. Simics-SystemC integration

While QEMU provides an interesting emulator example, it is not unique, and to prove our methodology is more widely applicable, we have also worked on adding SystemC capabilities to Virtutech's Simics [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.

We have developed a device to be plugged into Simics (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, because the 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. We use the same synchronization

management between the two simulators as described in this paper.

We tested this integration using a SystemC 16550 UART plugged into the Simics simulator. The integration was successful, with a very low performance penalty about 5% in a testbench between the SystemC UART and the Simics standard UART model.

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 in a later date). We accomplished partial results, enabling checkpointing for SC_METHODs only, and leaving the responsibility to identify what device variables are crucial in order for the device to be properly saved and restored to the user. More details of this work can be found in [13].

6. Wrapping QEMU

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 5).

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 pause and resume it in order to be able to yield (i.e. call `wait()`).

The original QEMU is based on an endless loop that performs all the necessary steps to do the virtual simulation. We broke this loop with a call to a synchronization function that computes the difference between QEMU time and SystemC time. This difference is the SystemC time caused by our wrapper, or a target, calling `wait()`. This is done to allow the simulator kernel to run others modules.

Our wrapper contains an SC_THREAD that runs the QEMU modified main loop and calls `wait()` when necessary. Also, the wrapper has two functions (`sc_read/sc_write`) to allow QEMU to use them to access to SystemC devices. These `sc_read/sc_write` use a TLM-2.0 socket in the same way we explained in section 3.

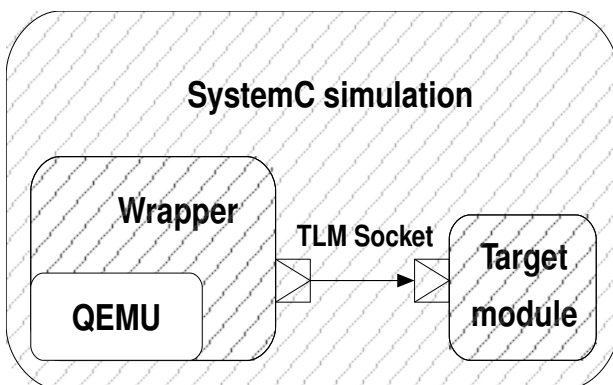


Figure 5: Wrapped QEMU simulation

The use of the same TLM-2.0 sockets in both implementations, allows to use the same SystemC devices in both simulations.

We have built the same testbench as we did for the first solution, with the difference being that now we have two SystemC modules (the QEMU Wrapper and the SystemC device) in the same hierarchical level (wich are connected).

Our preliminary tests don't show any performance difference with the previous proposed solution.

7. Future Work

Development of supporting temporal decoupling in our *SystemC bridge* is under development. This mechanism, allows a SystemC device to run ahead of the CPU emulators simulation time, in order to increase simulation speed and minimizing the context switching. This way, the device has its own local simulation time, and gets synchronized only when required, or after a fixed periodic time, managed by the quantum keeper. Currently we synchronize the device on a quantum calculated only from the CPU emulators notion of time, with no respect for the potential time indicated by a well behaved temporal decoupled SystemC model. Our plan is to adjust the quantum time we use by the time indicated by the SystemC model. As currently we synchronize both simulators at fixed periodic time, we think that we can support temporal decoupling with a minimum development on the *SystemC bridge*.

On the other hand, QEMU supports taking snapshots of the complete virtual machine state, including CPU state, memory, device state, etc. That state is saved to disk, and it can be restored at any time to get the emulated platform in the same state it was when the snapshot was taken. We plan to include mechanisms to support some checkpoint and restore to SystemC, and include this feature to our SystemC to QEMU bridge. This check pointing support in SystemC includes adding a mechanism to access internal device variables to save and restore them, and a mechanism to access and restore the event list from OSCI SystemC simulator. This work has been published by the authors in [13].

Last, the current QEMU version (0.10.5) supports KVM (Kernel Based Virtual Machine) [14]. This virtualization infrastructure supports native virtualization using some of the recent features of the Intel and AMD x86 processors.

Our very firsts trials with KVM show our *SystemC bridge* working without changes. QEMU globally presents and increased simulation speed as we would expected. We think that this new feature doesn't affect our work on linking SystemC and QEMU because KVM only affects the CPU emulator module. In short we believe that it will be possible to take advantage of the speed offered by KVM without changing the *SystemC bridge* mechanism. We intend to prove this.

In next phase of KVM development, KVM will support paravirtualization of real HW devices (current

work is in progress to support paravirtualization of Ethernet and block devices). We will investigate changes to our strategy to enable SystemC device models to still be used in such a framework.

Last, we plan to add this SystemC support (both solutions, systemC-bridge and SystemC wrapper) to more of the supported QEMU platforms, mainly MIPS and ARM. This step is straightforward because we only need a new virtual device for each of these platforms, as we did with the `sc_link` in the x86 platform

8. Conclusions

We have presented our work on adding SystemC capabilities to the QEMU platform emulator. We use TLM-2.0 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: making the OSCI simulator a slave of the QEMU emulator, and running SystemC simulation only when it is really needed.

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

Additionally, we have introduced a SystemC wrapper for QEMU that allows it to be present in a standard SystemC simulation as a TLM-2.0 Initiator. This wrapper allows designers to use standard SystemC based virtual platform tools to model the entire system, with no special requirements.

All source code and documentation is available at greensocs website (<http://wsw.greensocs.com>).

9. References

- [1] Virtutech website. <http://www.virtutech.com>
- [2] Imperas website. <http://www.ovpworld.org>
- [3] Coware website. <http://www.coware.com/products/virtualplatform.php>
- [4] Bellard, F. 2005. "QEMU, a fast and portable dynamic translator". In Proceedings of the Annual Conference on USENIX Annual Technical Conference (Anaheim, CA, April 10 - 15, 2005)..
- [5] Bochs website. <http://bochs.sourceforge.net/>
- [6] VMWare website. <http://vmware.com/products>
- [7] Microsoft Virtual PC 2007 website. <http://www.microsoft.com/windows/products/winfamily/virtualpc>
- [8] M. Montón, A. Portero, M. Moreno, B. Martínez, J. Carrabina. "Mixed SW/SystemC SoC Emulation Framework". IEEE International Symposium on Industrial Electronics (ISIE) 2007.
- [9] ForteDS website. <http://www.fortedds.com/products/cynthesizer.asp>
- [10] GreenSocs website. <http://www.greensocs.com/en/Projects/GreenSocket/>
- [11] ARM website. <http://www.arm.com/products/DevTools/VersatileFamily.html>
- [12] Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hällberg, Johan Högberg, Fredrik Larsson, Andreas Moestedt, Bengt Werner. "Simics: A Full System Simulation Platform", IEEE Computer, Feb 2002.
- [13] M. Montón, J. Engblom, M. Burton. "Checkpoint and Restore for SystemC Models". Proc. Forum on Design Languages (FDL), 2009.
- [14] KVM website. <http://www.linux-kvm.org>