On the Co-simulation of SystemC with QEMU and OVP Virtual Platforms

Alessandro Lonardi[1] and Graziano Pravadelli[1,2](✉)

[1] Department of Computer Science, University of Verona, Verona, Italy
alessandro.lonardi@univr.it
[2] EDALab s.r.l., Verona, Italy
graziano.pravadelli@edalab.it

Abstract. Virtual prototyping allows designers to set up an electronic system level software simulator of a full HW/SW platform to carry out SW development and HW design almost in parallel. To achieve the goal virtual prototyping tools allow the co-simulation between an efficient instruction set simulator, mainly based on dynamic binary translation of the target code, and simulation kernels for HW models, described by means of traditional hardware description languages, like, for example, SystemC. In this context, some approaches have been proposed for co-simulation between QEMU and SystemC, both from EDA companies and academic research groups. On the contrary, no paper addresses integration between Open Virtual Platform (OVP) and SystemC. Indeed, OVP models and the related simulator can be integrated into SystemC designs by using TLM 2.0 wrappers and opportune OVP APIs. However, this solution presents some disadvantages, like the incapability of supporting cycle-accurate models, and the necessity of re-design, in terms of SystemC modules, all OVP components that should be integrated in the target platform. To avoid such drawbacks, and provide an easy way to port SystemC models from a QEMU-based to an OVP-based virtual platform and vice versa, this paper presents a common co-simulation approach that works for integrating SystemC components with both QEMU and OVP. Experimental results show the effectiveness of the proposed architecture.

1 Introduction

Virtual prototyping is today an essential technology for modelling, verification and re-design of full HW/SW platforms [1]. With respect to the serialized approach, where the majority of SW is developed and verified after the completion of the silicon design, with the risk of failing aggressive time-to-market requests, virtual prototyping guarantees a faster development process by implementing the software part almost in parallel with the hardware design (Fig. 1). This enables software engineers to start implementation months before the hardware platform

This work has been partially supported by the EU large-scale integrating project CONTREX (FP7-2013-ICT-10-611146).

is complete. The core of virtual prototyping is represented by the virtual system prototype, i.e., an electronic system level (ESL) software simulator of the entire system, used first at the architectural level and then as a executable golden reference model throughout the design cycle. Virtual prototyping brings several benefits like, for example, efficient management of design complexity, decoupling of SW development from the availability of the actual HW implementation, and control of prototyping costs. More pragmatically, it enables developers to accurately and efficiently explore different solutions with the aim of balancing design functionality, flexibility, performance, power consumption, quality, ergonomics, schedule and cost.

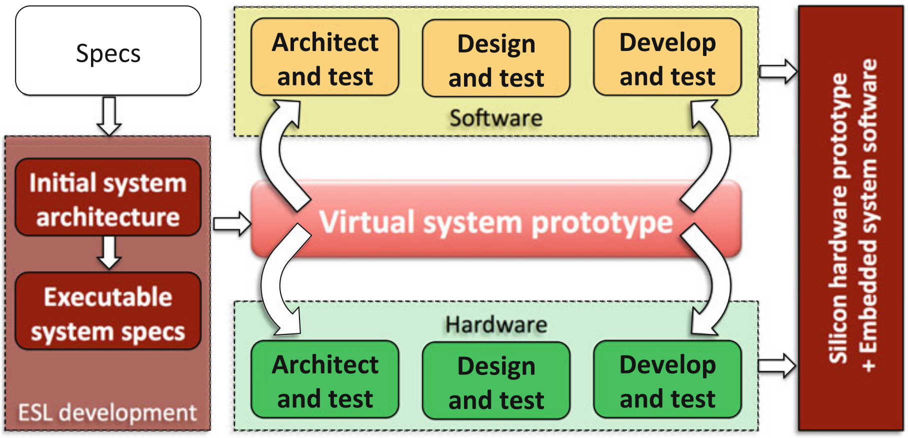

Fig. 1. The virtual prototyping approach.

A common aspect in modern virtual platform approaches is the use of an ISS that exploits DBT [2]. This technique has become the de facto standard to guarantee high speed and accuracy of cross-compiled software, thus many groups, both in industry and academia, have started to consider DBT as a key technology of their virtual prototyping solutions. Companies, like VaST (acquired by Synopsys) and Virtutech (acquired by Intel/Windriver) introduced virtual prototyping platforms for rapid development and fast execution of target software [3,4]. Meanwhile all of the three main EDA companies (Synopsys, Cadence, Mentor) have their own virtual prototyping platform [5–7]. All of them are using fast simulation technologies that are based on DBT. However, as they target to keep their markets, most of them did not support open standard interfaces. The technology itself is proprietary and provided under binary format, making it impossible to perform modifications within the models, i.e., addition of new features or annotations. Furthermore, most commercial products bind the customer to their own standard languages and features limiting the reuse of models towards and from other tools.

Taking an opposite view to this closed-source approach, several research groups have recently focused on QEMU [8], an open-source DBT-based virtualization platform that allows cross-platform operating system execution and provides a very fast emulation comparable to commercially available tools. As an alternative to QEMU, Imperas has proposed the Open Virtual Platform (OVP) initiative [9]. OVP, similarly to QEMU, offers a simulation kernel based on a code morphing DBT mechanism to guarantee very efficient simulation of full HW/SW systems. It includes several models of processors, peripherals and behavioural components that the software communicates with. One difference between QEMU and OVP is that the former is more targeted for *homogeneous* single/multi-core platforms, while OVP has been thought to better support *heterogeneous* MPSoC architectures. For this reason, it is likely that OVP will gain more and more consensus, in the near future, for virtual prototyping of modern cyber-physical systems, which heavily rely on heterogeneous architectures.

The advent of QEMU and OVP, due to their high efficiency and flexibility, has then replaced SystemC as the main open-source approach for virtual prototyping of full HW/SW systems. SystemC allows to model entire systems at different levels of abstractions and with different degrees of detail, however, its simulation performances are poor when accurate and realistic CPU models, good enough to run real operating systems, are desired. SystemC processes are executed sequentially and managed by a centralized kernel, which definitely represents a bottleneck for simulation. Thus, for example, the processing power offered by multi- and many-core architectures cannot be sufficiently exploited by adopting SystemC. However, SystemC guarantees to model customized (non standard) HW components with a level of details that cannot be achieved neither by QEMU nor by OVP. Furthermore, effective virtual prototyping approaches cannot exclude design reuse and bottom-up component-based design, where already defined (SystemC) models are integrated in new prototypes, to reduce the time to market. For this reason, several approaches, like [10–13], propose the use of QEMU, for efficient instruction set simulation, in combination with SystemC, for specification and reuse of customized HW components. In particular, [10] proposes a QEMU-SystemC approach developed by the GreenSoCs company which represents the main way for connecting SystemC models, as peripherals, to QEMU. On the contrary, Imperas supports the integration between SystemC and OVP by allowing the encapsulation of OVP models, wrapped by a TLM 2.0 interface, inside SystemC designs. However, such an encapsulation presents some disadvantages. First, it works natively only for TLM designs; the OVP user guide discourages from the integration of OVP models into non-TLM systems, highlighting the risk of incorrect results [14]. This prevents accurate simulation of RTL SystemC models. Secondly, encapsulation of OVP models inside SystemC modules is a quite complex operation, which does not allow a rapid reuse of SystemC components. In fact, all OVP models included in the target platform must be redesigned in terms of SystemC modules (through OVP APIs and TLM interfaces) to be co-simulated with the customized SystemC hardware. Furthermore, the porting of a SystemC component, originally linked inside a

QEMU-based platform, towards an MPSoC OVP-based platform would require a huge effort. Similar considerations apply in the opposite case, when porting from the OVP world to QEMU's one is desired.

To overcome these drawbacks, this paper, as an extension of [15], proposes a common architecture to integrate SystemC components in both QEMU and OVP-based virtual platforms. Starting from the idea presented in [10], our approach defines an efficient, shared-memory based architecture that allows the communication between a SystemC peripheral and a SW program, which runs either on a QEMU or on an OVP CPU model, through a common PCI virtual device bridge. In this way, porting of SystemC components from a QEMU-based to an OVP-based virtual platform is straightforward, since it requires only to redefine the virtual device in terms of either OVP or QEMU APIs. Finally, there is no limitation about the abstraction level at which SystemC models are implemented.

The rest of the paper is organized as follows. Section 2 briefly presents the main characteristics of QEMU and OVP, and it summarizes approaches for their co-simulation with SystemC. Section 3 describes the proposed common co-simulation architecture. Section 4 is devoted to experimental results. Finally, concluding remarks are reported in Sect. 5.

2 Background and Related Works

QEMU and OVP are two very popular open-source environments for rapid prototyping of virtual platforms. Both of them allow a very efficient emulation of several architectures and they can interface with many types of physical host hardware. QEMU is mainly intended for simulation of a fixed, defined single processor platform. OVP is more suited to model heterogeneous platforms with arbitrary shared and local memory configurations. Next subsections report a brief summary of their main characteristics and the state of the art related to their integration with SystemC.

2.1 QEMU

QEMU is a machine emulator relying on dynamic binary translation of the target CPU application code. Each instruction of the target CPU is translated into a set of micro-operations that are implemented by C functions for the host machine. Such C functions are then compiled to obtain a dynamic code generator which is called, at run time, to generate the corresponding code to be executed on the host machine. QEMU works at basic block level. Each block is translated, the first time it is encountered, into the corresponding code for the target CPU, then it is stored in a cache for future uses. In addition to such a common feature, different optimizations can be implemented to further keep execution speed close to native execution, like, for example, dynamic recompilation where some parts of the code are recompiled to exploit information that are available only at run time.

There is no a native way to integrate SystemC models into QEMU, thus some co-simulation approaches have been proposed in the past [10,12,13,16]. The intent of [10] is to facilitate the development of software and device drivers for whatever operating system without spending too much effort on modifying the virtual platform itself by plugging SystemC TLM 2.0 models into the QEMU-based virtual platform. This work is further extended in [13] where the authors introduce a checkpoint-based feature to save and restore the SystemC state into the simulator. Alternatively, in [12,16], the authors propose a QEMU/SystemC-based framework for virtual platform prototyping that cannot only estimate the performance of a target system, but also co-simulate with hardware models down to the cycle accurate level.

2.2 OVP

OVP is a virtual platform emulator released by Imperas for enabling the simulation of embedded systems running real application code. It includes a fast Just-In-Time (JIT) code morphing simulator engine (OVPsim), a set of APIs for modelling new platforms and components, like CPUs and peripherals, and a set of already designed models. An OVP platform is composed of one or more processors, memories, interconnections, and possibly some peripherals. A platform is modelled through the following sets of APIs:

- Innovative CPU Manager (ICM): ICM functions enable instantiation, interconnection and simulation of complex multiprocessor platforms composed of processors, memories and busses in arbitrary topology.
- Virtual Machine Interface (VMI): VMI functions are used to create new models of processors to be run inside the OVPsim. CPU instructions are mapped onto the JIT code morphing compiler primitives to speed-up the simulation.
- Behavioural Hardware Modelling (BHM): BHM functions allow to define behavioural models of hardware and software components that act as peripherals to the processors. They are executed by the Peripheral Simulation Engine (PSE) in a separate (protected) process with respect to OVPsim.
- Peripheral Programming Model (PPM): PPM functions are used in conjunction with BHM functions to model interfaces to the platform and connections (bus ports, net ports, etc.).

The use of OVP to model heterogeneous multiprocessor architectures is described in [17] through a set of case studies. A drag and drop interactive approach for MPSoC exploration using OVP is proposed in [18]. A technique exploiting OVP for development and optimization of embedded computing applications by handling heterogeneity at the chip, node, and network level is proposed in [19]. Heterogeneity is handled by providing an infrastructure to connect multiple virtual platforms like OVP and QEMU.

None of previous works considers co-simulation between an OVP-based virtual platform and external SystemC models. However, OVPsim platform models can be compiled as shared objects. Thus, they can be encapsulated in any simulation environment that is able to load shared objects, including SystemC. The

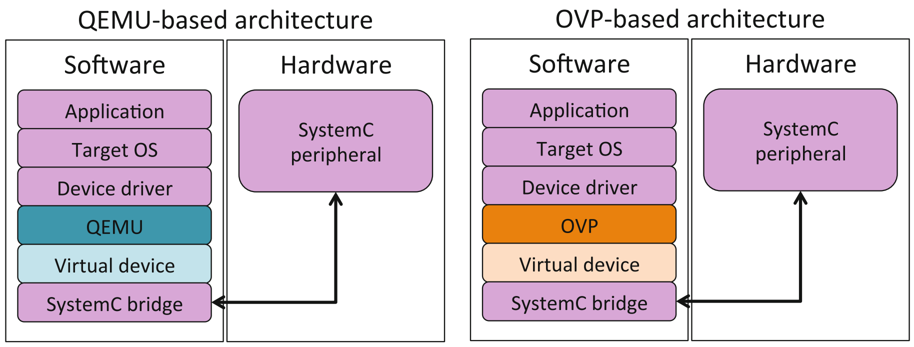

Fig. 2. Co-simulation architectures. Pink boxes indicate unchanged code between QEMU and OVP (Colour figure online).

integration with SystemC is enabled by the ICM APIs. OVP models and OVPsim can be encapsulated inside a SystemC design to create a virtual platform where SystemC components and OVP models simulate together under the control of the SystemC simulation engine. However, the OVP CPUManager is not intended for cycle-accurate or pin-level simulation. For this reason ICM APIs provide only loosely timed TLM 2.0 interfaces. Thus, integration of OVP models in a SystemC RTL system would require the definition of TLM-to-RTL transactors. However, the OVP user guide discourages from the integration of OVP models into non-TLM systems, highlighting the risk of incorrect results [14]. Moreover, as shown in our experimental results, simulation performances are heavily penalized when OVP is integrated inside SystemC.

3 Co-simulation Architecture

In this paper, the HW/SW co-simulation architecture depicted in Fig. 2 is proposed. Similarly to the approaches presented in [10,13], it reflects the traditional operating system-based stack where software applications interact with hardware peripherals through device drivers. The target platform, where software applications run, is emulated by using, indifferently, QEMU or OVP, and it is connected to a SystemC hardware peripheral through a *virtual PCI*[1] *device, a SystemC bridge* and a device driver for the target operating system.

The virtual PCI device, connected to the PCI bus of the virtual platform, acts as an interface between the SystemC simulator and the QEMU or OVP virtual platforms. The virtual device code is different for the QEMU-based and the OVP-based architectures, since it depends on the APIs exported by QEMU and OVP for modeling new devices.

The SystemC bridge consists of a set of functions that allow the communication with SystemC. The bridge is compiled as a C library linked to the

[1] The same approach can be adopted also for other kinds of buses, like for example AMBA.

implementation code of each virtual device, thus it is independent from the selected virtual platform and it is the same for both the QEMU-based and the OVP-based architectures.

Finally, a device driver must be developed for the target operating system to use the hardware peripheral. Its code is clearly independent from the selected virtual platform and it does not need to be changed when it is moved to the actual platform.

This approach allows a rapid interchange from a QEMU-based to an OVP-based SystemC co-simulation and vice versa, since only the virtual device must be re-coded moving from a QEMU virtual platform to an OVP virtual platform.

Further details about the SystemC bridge and the virtual device are reported, respectively in Sects. 3.1 and 3.2, while Sect. 3.3 describes how the device driver and the virtual device interact to implement the interrupt handling mechanism.

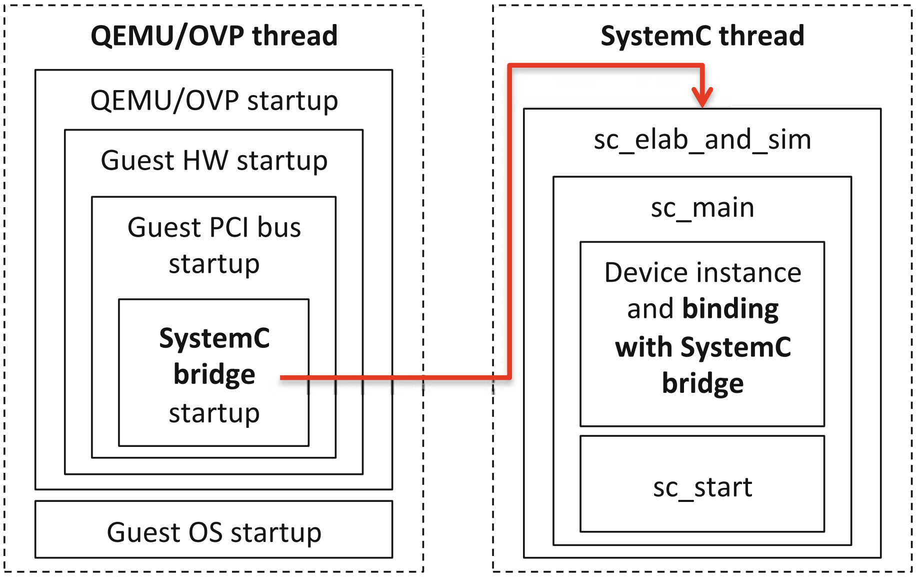

Fig. 3. Startup of the cosimulation between QEMU/OVP and SystemC.

3.1 SystemC Bridge

The SystemC bridge is a C++ class implementing a singleton design pattern that exposes a set of APIs towards the virtual device for interfacing with SystemC. Moreover, it manages the communication protocol and the synchronization mechanism between QEMU/OVP and SystemC. Since both QEMU and OVP are written in C, the bridge wraps the SystemC APIs through a set of C functions included in a library, which is statically linked to the SystemC runtime library.

The QEMU and the OVP virtual platforms[2] initially call a function implemented in the bridge to start the SystemC simulator. Until the SystemC runtime is operative, the virtual platform is blocked to prevent its premature request to the hardware device. Differently from [10], the SystemC simulator is run as a separate thread inside the same process where the QEMU emulator or OVP simulator is executed (Fig. 3), such that the communication between the two worlds is based on shared memory and thread synchronization primitives. This prevents the use of expensive interprocess communication mechanisms (like sockets). Then, the starting routine of the thread launches the SystemC *sc_main* function where the following steps are executed:

- instantiation and initialization of the SystemC device to be connected to the bridge; in particular, input and output ports of the device are registered in the bridge;
- unlocking of the semaphore that is blocking the virtual platform;
- starting of the SystemC simulator.

The SystemC bridge exports two functions (i.e., *sc_ioport_read* and *sc_ioport_write*) towards the virtual platform to allow reading from/writing to the SystemC module.

Read operations are performed by calling the *sc_ioport_read* function. This invokes the *read* method of the SystemC bridge on the target signal (*Systemc_To_VirtualPlatform_Signal*), which represents an output port for the SystemC device. The signal is implemented like a proxy for a *sc_core::sc_out* as reported in Fig. 4. In particular, it is an *SC_MODULE* with a method, *run*, which updates the signal value each time a change happens in the output port of the SystemC device. Then, the *read* method retrieves the current value of the signal each time the virtual platform calls the *sc_ioport_read* function, as reported in the sequence diagram of Fig. 5. To guarantee atomicity of operations, *read* and *run* methods are made mutually exclusive through the use of a mutex.

Write operations are performed in a similar way by invoking the *sc_ioport_write* function. As shown in the sequence diagram of Fig. 7, it calls the *write* method of the SystemC bridge on the target signal (*VirtualPlatform_To_Systemc_Signal*), which represents an input port for the SystemC device. Such a signal is implemented in a similar way with respect to the *Systemc_To_VirtualPlatform_Signal* (Fig. 6). The only difference is represented by the necessity of preventing concurrent write operations by the virtual platform, which would be missed by the SystemC runtime. This is obtained by using a mutex that is locked as soon as the *write* method is invoked and by a flag that is used to notify a pending write operation to the SystemC device. Then, the *run* method is executed at each clock cycle checking for a pending operation from the virtual platform; in case of its presence, the pending value is written to the

[2] In the following, we use the generic name *virtual platform* to refer, without distinction, to the QEMU as well as the OVP environment.

```
template <typename T>
  SC_MODULE(Systemc_To_VirtualPlatform_Signal),
   public Systemc_To_VirtualPlatform_Signal_Base {

  SC_HAS_PROCESS(SystemC_To_VirtualPlatform_Signal);

   Systemc_To_VirtualPlatform_Signal(
     sc_core::sc_out<T>& port,
     sc_core::sc_module_name name) : sc_module(name) {

     SC_METHOD(run);
     sensitive << signal;
     port.bind(signal);
   }

   virtual uint64_t read() {
     Scoped_Lock sl(mutex);
     return static_cast<uint64_t>(native);
   }

   void run() {
     Scoped_Lock sl(mutex);
     native = signal.read();
   }

   virtual std::string name() const {
     return sc_module::name();
   }

   private:
     Mutex mutex;
     T native;
     sc_core::sc_signal<T> signal;
};
```

Fig. 4. Implementation of a SystemC to virtual platform signal to allow QEMU/OVP reading from the SystemC device.

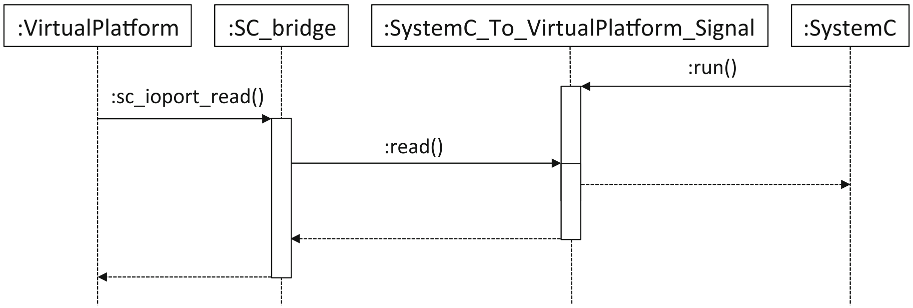

Fig. 5. Sequence diagram of a read operation from a SystemC device.

port of the SystemC device and the mutex is unlocked allowing further write operations from the virtual platform.

To optimize performance, time synchronization between QEMU/OVP and SystemC is not clock accurate, but it happens only when a call to a SystemC device is performed in order to update the SystemC time to be the same as

```
template <typename T>
  SC_MODULE(VirtualPlatform_To_Systemc_Signal),
  public VirtualPlatform_To_Systemc_Signal_Base {

  SC_HAS_PROCESS(VirtualPlatform_To_Systemc_Signal);

  VirtualPlatform_To_Systemc_Signal(
    sc_core::sc_in<T>& port,
    sc_core::sc_clock& clock_signal,
    sc_core::sc_module_name name) : sc_module(name) {

        is_vp_write = false;
        SC_METHOD(run);
        sensitive_pos << clock.pos();
        clock.bind(clock_signal);
        port.bind(signal);
  }

  virtual void write(uint64_t value) {
    token.hold;
    native = static_cast<T>(value);
    is_vp_write = true;
  }

  void run() {
    if (is_vp_write) {
      signal.write(native);
      is_vp_write = false;
      token.release();
    }
  }

  virtual std::string name() const {
    return sc_module::name();
  }

  private:
    Token token;
    T native;
    sc_core::sc_signal<T> signal;
    bool is_vp_write;
    sc_core::sc_in_clk clock;
};
```

Fig. 6. Implementation of a virtual platform to SystemC signal to allow QEMU/OVP writing to the SystemC device.

QEMU time. This only guarantees that two subsequent operations do not interfere with each other, which is a sufficient condition for functional verification but not enough for other kind of analysis (e.g., power consumption/timing estimation). Future works will deal with a more accurate time synchronization.

3.2 Virtual Device

A device can be made visible to the QEMU and OVP virtual platforms by implementing a virtual device. In our work, we considered PCI devices, however, in a similar way other kinds of bus can be adopted. Differently from the SystemC bridge, whose implementation code is the same for QEMU and OVP, the creation

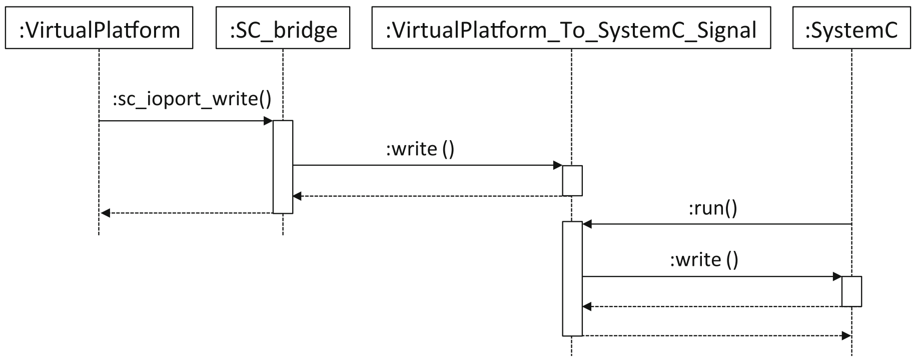

Fig. 7. Sequence diagram of a write operation to a SystemC device.

of a virtual device is tightly coupled to the APIs exported by the virtual platform. Thus, QEMU and OVP implementations will be described separately.

QEMU Virtual Device. The implementation of the QEMU virtual device is based on the PCI APIs. The device is plugged to the selected machine (in our experiments a Malta platform) and connected to a PCI bus. The device characteristics (name, parent class, size of the occupied memory, initialization routine) are described into a structure which is passed to the selected machine during the device registration phase. The initialization routine is executed when a new instance of the device is initialized. Its role consists of registering the I/O ports memory regions and starting the SystemC simulation through the SystemC bridge. The registration of the I/O ports memory region reserves a chunk of memory for the virtual device and retrieves the pointers to functions for reading/writing from/to the virtual device I/O ports that are mapped to the actual SystemC device I/O ports. Such function invokes the corresponding *sc_ioport_read* and *sc_ioport_write* of the SystemC bridge.

OVP Virtual Device. The implementation of the virtual device in OVP is split in two parts: the device and the intercepted functions. The device consists of a C application with a standard *main* function. It first executes a set of initialization activities (SystemC simulation, PCI configuration header, PCI memory regions); then it connects the PCI configuration port (necessary to read the PCI configuration header) and the PCI master bus. Finally, it registers some call back functions that are triggered at each read/write operation from/to the PCI I/O port regions. The interaction between the SystemC bridge and the virtual device is performed by means of a set of intercepted functions. The PSE simulator intercepts such functions through the Application Binary Interface (ABI), which specifies size, layout and alignment of data types, how an application should make a system call to the operating system, and the calling conventions (how arguments of a function are passed, and how return value is retrieved). In particular, there is an intercepted function for reading from the SystemC device and one for writing to the SystemC device. Their role consists in calling the corresponding *sc_ioport_read* and *sc_ioport_write* of the SystemC bridge.

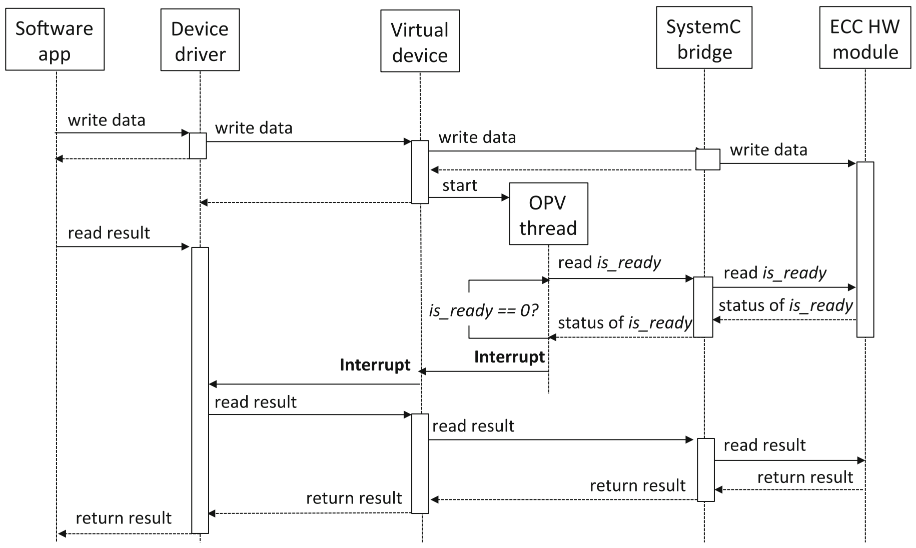

Fig. 8. Sequence diagram of the interrupt mechanism for ECC benchmark.

3.3 Interrupt Handling

The basic architecture described in the previous sections supports only I/O requests from the virtual platform to the SystemC device, while the second acts only as a slave. However, an effective virtual prototyping solution cannot forget to provide support for interrupt-based asynchronous communication, through which, for example, a device can notify the completion of a task to the CPU. To achieve this goal, an interrupt handling mechanism has been defined. Its implementation is composed of two parts, which does not involve the SystemC bridge: the first part is embedded in the device driver, the second in the virtual device. As an explanatory example, we refer to the interaction between a SW application and a SystemC module representing and error correction code (ECC) that we used in the experimental result section (see Sect. 4). Figure 8 shows the sequence diagram that describes these interactions. In particular, a write operation is performed on the ECC module and then a subsequent read operation is called. The device driver and the virtual device manage the correct sequencing of the two operations such that the *read* is executed only after the *write* has completed. The completion of the *write* is notified by the virtual device through an interrupt that is raised as soon as the *is_ready* line of ECC becomes high.

```
// variable declarations

// semaphore for unlocking read operation upon interrupt
struct semaphore readLock;

// device driver opening
static int open(struct inode* inode, struct file* file) { ... }

// device driver releasing
static int release(struct inode* inode, struct file* file) { ... }

// read operation callback
static ssize_t read(struct file* file, char* buf,
    size_t count, loff_t* ppos)
{
  // initializations and sanity checks
  ...
  // blocking via semaphore (unlocked by interrupt handler method)
  wait(&readLock);
  // read operation
  ...
}

// write operation callback
static ssize_t write(struct file* file, const char* buf,
    size_t count, loff_t* ppos)
{
  // write operation
  ...
}


// Interrupt handler method
static irqreturn_t irq_handler(int received_irq, void* dev_id) {
    // data ready unlock semaphore
    signal(&readLock);
    return IRQ_HANDLED;
}

// Device driver initialization
static int __init init(void)
{
    // variables and utilities initializations
    ...

    // Semaphore initalization and interrupt handler registration
    sema_init(&readLock, 1);
    if((err = request_irq(15, irq_handler,
        IRQF_SHARED, DRV_NAME, (void*)(irq_handler))))
    {
        pr_err(''Cannot obtain irq, aborting'');
        return -1;
    }
        return 0;
}

// other functions
...
```

Fig. 9. Device driver implementation for ECC.

```
// Interrupt checking thread method
static void interruptCheckThread(void* data)
{
  Uns32 result = 0;

  while(1)
  {
    // wait until write operation is called on the device\\
    bhmWaitEvent(threadEventHandle);
    while(result == 0)  // Check if ECC is ready?
    {
      result = read_ecc_is_ready();
      // yield for a while to avoid hogging the CPU
      bhmWaitDelay(100);
    }

    // When ECC is ready send interrupt
    ppmWriteNet(intPort, 1);
    ppmWriteNet(intPort, 0);
  }
}

// main: used as OVP peripheral constructor
int main(int argc, char **argv) {
  // peripheral initializations
  ...
  // PCI initialization
  pciHeaderInit();
  pciMappingInit();
  pciRegisterCallBack();
  // Bus ports initializations
  ppmOpenSlaveBusPort(``config'', config_window, sizeof(config_window));
  pciConnectConfigSlavePort(PREFIX, NULL);

  // initialization of the Interrupt line
  intPort = ppmOpenNetPort(``sclinkInterrupt'');

  // initialization of the interrupt checking thread
  threadEventHandle = bhmCreateEvent();
  threadHandle = bhmCreateThread(interruptCheckThread,
                          NULL, threadName, &threadStackData[THREAD_STACK]);
  bhmEventHandle finished = bhmGetSystemEvent(BHM_SE_END_OF_SIMULATION);
  bhmWaitEvent(finished);
  bhmMessage(``I'', PREFIX, ``Shutting down'');

  terminate();
  return 0;
}
```

Fig. 10. Interrupt handling inside the OVP virtual device for ECC.

Monitoring such a line is the role of a dedicated OVP thread, which is activated as soon as a new *write* is requested by the software application.

The device driver side is implemented in the traditional way. From the point of view of interrupt handling, it just requires the definition of a function that is called through the interrupt service routine when this is triggered by the arrival of the corresponding interrupt. For example, let us consider the piece of code reported in Fig. 9. It shows the skeleton of the device driver written for ECC. An interrupt is raised by the ECC module as soon as a write operation is concluded. As a consequence, the *irq_handler* function is executed (lines 34–38), which unlocks the *readLock* semaphore (line 36). Such a semaphore is used to block the *read* routine that waits till the ECC device is ready for a read operation (line 19).

Then, the most effort for implementing the interrupt handling resides in the virtual device side. The mechanism is the same for both QEMU and OVP, given that, on the virtual device side, either QEMU or OVP APIs are adopted. Figure 10 shows the skeleton for the OVP case. ECC raises an interrupt by fixing to 1 the line *is_ready* as soon as a write operation on the device is completed. At platform level, a signal is used to connect the virtual device to the interrupt lines provided by the platform chipset (line 36). When an interrupt is generated by the SystemC device, the interrupt is notified to the chipset through such a signal (lines 18–19). To catch the interrupt from the SystemC module, the virtual device creates an OVP thread (line 40–41) that cyclically checks the *is_ready* line of the SystemC module (lines 1–21). A waiting time is introduced between two consecutive checks to avoid hogging the CPU (line 14). The thread is finally suspended, to reduce its busy waiting activity, after the interrupt is raised till a new write operation is called on the ECC (line 9).

4 Experimental Results

Experimental results have been executed by setting up a virtual platform composed of a MIPS-based Malta platform and a SystemC RTL module connected through the PCI bus. The SystemC module implements an Error Correction Code (ECC) algorithm as an external hardware peripheral. Two versions of the platform have been implemented, one based on QEMU and one on OVP. The Malta platform is equipped with a MIPS 34Kf CPU running a Debian 6.0.9 with a 2.6.32.5 Linux kernel. The host machine is an Intel Core2Quad Q6600 @2.4 GHz with 4 GB of RAM, running Ubuntu 12.04, QEMU 1.0.50, OVP v20140127.0 and SystemC 2.3.0.

Experimental results concerning simulation times are reported in Table 1. A software application running on the target CPU has been required to ask the SystemC peripheral to compute the ECC a variable number of times (Column ITERATIONS. The experiment has been executed on the QEMU-SystemC and OVP-SystemC common architecture described in Sect. 3 (respectively, columns QEMU-SYSTEMC BRIDGE and OVP-SYSTEMC BRIDGE). Moreover, a virtual platform has been implemented by wrapping the OVP modules of the Malta platform into a SystemC design by following the official guidelines reported in the OVP documentation [14] (columns OVP WRAPPED INTO SYSTEMC). This requires also to implement a transactor to convert the RTL interface of the ECC module towards a TLM 2.0 interface. Two different sets of experiments have been executed by setting the clock period of the SystemC RTL peripheral, respectively to 10 ms (left part of the table) and 1 ns (right part of the table). This different setting negatively impacted on the results of the official OVP-SystemC schema, since it was not able to carry on the simulation for clock periods lower than 10 ms. The simulation blocked during the initialization phase. Indeed, OVP guidelines explicitly state that CPU manager is based on the TLM

2.0 loosely time model, and it is not intended for cycle-accurate simulation of RTL components. Attempting to use other models gives incorrect results. On the contrary, with a period of 1 ms the simulation works correctly, but the simulation time is negatively affected with respect to the co-simulation schemas proposed in this paper.

Table 1. Comparison of simulation times between the SystemC-QEMU/OVP common co-simulation architecture proposed in this paper and the native way of integrating OVP into SystemC described in the OVP guidelines [14].

Iterations	SystemC clock set at 10 ms			SystemC clock set at 1 ns		
	Boot	Boot & run	Run	Boot	Boot & run	Run
QEMU-SystemC bridge						
5,000	80.54 s	92.84 s	12.30 s	79.79 s	91.18 s	11.39 s
10,000		103.11 s	22.57 s		102.45 s	22.66 s
50,000		191.07 s	110.53 s		197.65 s	117.86 s
100,000		314.65 s	234.11 s		305.84 s	226.05 s
OVP-SystemC bridge						
5,000	85.50 s	125.07 s	39.57 s	79.95 s	123.68 s	43.73 s
10,000		170.19 s	84.69 s		171.41 s	91.46 s
50,000		475.60 s	390.10 s		474.30 s	394.43 s
100,000		899.92 s	814.42 s		903.76 s	823.81 s
OVP wrapped into SystemC [14]						
5,000	101.83 s	458.46 s	356.63 s	-	-	-
10,000		835.62 s	733.79 s	-	-	-
50,000		3808.38 s	3706.55 s	-	-	-
100,000		7509.38 s	7407.55 s	-	-	-

The table reports the time required to boot and shut down the virtual platform without running the application (Boot)[3], the time required to boot, run the application and shut down (Boot & run), and finally, the time referred to only the run of the application (Run). This final value is the most interesting, since it includes the time required for the communication between the QEMU/OVP virtual platform and the SystemC peripheral. There is a significant difference in the communication time. This is highlighted in Table 2, where a comparison between the OVP-SystemC co-simulation proposed in this paper and the native way of integrating OVP into SystemC described in the OVP guidelines is reported, taking the QEMU-SystemC architecture, which is the fastest, as reference for the simulation time. Columns of the table report the simulation

[3] As expected, the boot time is not influenced by the iteration number in all the three schemas.

Table 2. Comparison between the OVP-SystemC co-simulation proposed in this paper and the native way of integrating OVP into SystemC described in the OVP guidelines [14], taking the QEMU-SystemC architecture as reference for the simulation time.

Iteration	SystemC clock set at 10 ms			SystemC clock set at 1 ns		
	QEMU-SC Sim. time	OVP-SC Mul. fac.	OVPW-SC Mul. fac.	QEMU-SC Sim. time	OVP-SC Mul. fac.	OVPW-SC Mul. fact
Boot						
5,000	80.54 s	1.1x	1.3x	79.79 s	1.0x	-
10,000						
50,000						
100,000						
Run						
5,000	12.30 s	3.2x	29.0x	11.39 s	3.8x	-
10,000	22.57 s	3.8x	32.5x	22.66 s	4.0x	-
50,000	110.53 s	3.5x	33.x5	117.86 s	3.3x	-
100,000	234.11 s	3.5x	31.6x	226.05 s	3.6x	-

time for the QEMU-based architecture (QEMU-SC Sim. time), and its ratio (multiplication factor) with respect to the simulation time of the OVP-SystemC archicture based on the SytemC bridge (OVP-SC Mul. fac.) and the architecture proposed in [14] (OVPW-SC Mul. fac.). While boot time (columns Boot) is almost similar among the three approaches, the communication with the SystemC peripheral introduces an higher overhead in the OVP-based platforms rather than in the QEMU one (columns Run). However, it is worth noting that the overhead is much higher by adopting the co-simulation proposed in [14] with respect to using the bridge-based architecture proposed in this paper. This shows that the architecture described in this paper is more efficient with respect to wrapping OVP models into SystemC.

5 Conclusions

In this paper, we present a common architecture to integrate SystemC with both QEMU and OVP. The architecture is based on a SystemC bridge, which manages the communication protocol and the synchronization mechanism between QEMU/OVP and SystemC, and a virtual device, which acts as an interface between the SystemC simulator and the QEMU or OVP world. Interrupt handling is also supported. The bridge is the same for both QEMU and OVP, while only the virtual device must be coded according to the APIs exported by QEMU and OVP. This allows to rapidly reuse SystemC components from a QEMU-based to an OVP-based virtual platform and vice versa.

Experimental results highlighted that the QEMU-based approach is almost 4 times faster than the corresponding OVP-based approach. However, compared to the official way of integrating SystemC with OVP reported in the OVP guidelines, the OVP co-simulation approach proposed in this paper is one order of magnitude faster. Furthermore, we support cycle-accurate simulation of RTL (as well as TLM) models, while only TLM 2.0 loosely timed models work properly with the official OVP-SystemC co-simulation schema.

Future works will be mainly devoted to the definition of a more precise timing synchronization. Currently, synchronization between QEMU/OVP and SystemC is not clock accurate. A further extension will be related to the possibility of instantiating an arbitrary number of SystemC components. In the current approach, this requires to run a separate instance of the SystemC simulation kernel for each SystemC model.

Acknowledgements. The authors would like to thank Filippo Cucchetto and Stefano Angeleri for their contribution in applying the proposed architecture to the case study reported in experimental results.

References

1. Hellestrand, G.: Enhance communications platform design with virtual systems prototyping. http://www.embeddedintel.com/special_features.php?article=115
2. Ebcioglu, K., Altman, E., Gschwind, M., Sathaye, S.: Dynamic binary translation and optimization. IEEE TCOMP **50**(6), 529–548 (2001)
3. VaST's new CoMET 5 systems engineering environment for architectural design and exploration avoids chip respins and speeds development. http://www.businesswire.com/news/home/20040524005033/en/VaSTs-CoMET-5-Systems-Engineering-Environment-Architectural.VNKYKsbS5U8
4. Wind River Simics. http://www.windriver.com/products/simics/simics_po_0520.pdf
5. Synopsys, CoMET-METeor. http://www.synopsys.com/Systems/VirtualPrototyping/Pages/CoMET-METeor.aspx
6. Graphics, M..: Vista virtual prototyping. http://www.mentor.com/esl/vista/virtual-prototyping/
7. Cadence, Virtual system platform. http://www.cadence.com/products/sd/virtual_system/
8. Bellard, F.: QEMU, a fast and portable dynamic translator. In: Proceedings of USENIX ATEC, pp. 41–46 (2005)
9. http://www.ovpworld.org/
10. Monton, M., Carrabina, J., Burton, M.: Mixed simulation kernels for high performance virtual platforms. In: ECSI FDL, pp. 1–6 (2009)
11. Becker, M., Di Guglielmo, G., Fummi, F., Mueller, W., Pravadelli, G., Xie, T.: RTOS-aware refinement for TLM2.0-based HW/SW designs. In: Proceedings of ACM/IEEE DATE, pp. 1053–1058 (2010)
12. Chiang, M.-C., Yeh, T.-C., Tseng, G.-F.: A QEMU and SystemC-based cycle-accurate ISS for performance estimation on SoC development. IEEE TCAD **30**(4), 593–606 (2011)
13. Monton, M., Engblom, J., Burton, M.: Checkpointing for virtual platforms and SystemC-TLM. IEEE TVLSI **21**(1), 133–141 (2013)

14. Using OVP models in SystemC TLM2.0 platforms (2013). http://www.ovpworld.org/using-ovp-models-with-osci-systemc-tlm20-platforms-to-gain-200-500-mips-performance
15. Cucchetto, F., Lonardi, A., Pravadelli, G.: A common architecture for co-simulation of SystemC models in QEMU and OVP virtual platforms. In: Proceedings of IEEE VLSI-SOC, pp. 1–6 (2014)
16. Yeh, T.-C., Chiang, M.-C.: On the interface between QEMU and SystemC for hardware modeling. In: Proceedings of IEEE ISNE, pp. 73–76 (2010)
17. Rekik, W., Ben Said, M., Ben Amor, N., Abid, M.: Virtual prototyping of multiprocessor architectures using the open virtual platform. In: Proceedings of IEEE ICCAT, pp. 1–6 (2013)
18. Marchesan Almeida, G., Bellaver Longhi, O., Bruckschloegl, T., Hubner, M., Hessel, F., Becker, J.: Simplify: a framework for enabling fast functional/behavioral validation of multiprocessor architectures in the cloud. In: Proceedings of IEEE IPDPSW, pp. 2200–2205 (2013)
19. Jung, Y., Park, J., Petracca, M., Carloni, L.: netShip: a networked virtual platform for large-scale heterogeneous distributed embedded systems. In: Proceedings of ACM/EDAC/IEEE DAC, pp. 1–10 (2013)