Checkpointing for Virtual Platforms and SystemC-TLM

Màrius Montón, Jakob Engblom, and Mark Burton

***Abstract*—Integrating simulation models created using different simulation systems is a common problem when constructing virtual platforms. Different companies and different departments can create models, and virtual platforms for different purposes using different tools. There are also existing models that need to be integrated into new tools, or the other way around. The simulators can be quite different in details, even in the case of transaction-level models. We present work in integrating SystemC transaction-level models into two typical full-system simulation environments, QEMU and Simics. We present issues in reconciling the semantics of the different platforms, and our proposed solutions. In the Simics integration, we additionally enable checkpointing in the models, based on the Simics checkpoint mechanism.**

***Index Terms*—SystemC, system-level verification, transaction-level modeling, virtual prototyping.**

I. Introduction

A VIRTUAL platform (VP) or full-system simulation is a simulation of an existing or future hardware platform, built to facilitate software development for the platform. VPs normally run unmodified operating systems and software stacks that also run on the physical platform. To run the same software as the (eventual) physical hardware, the VP has to model (no more than) the behavior of the hardware as the software will interact with it.

Virtual platforms for software development are usually built by using transaction-level modeling (TLM). TLM provides an abstraction for the communication between components of a system, performing communications actions like a memory read or write or network packet transmission as a single step rather than as a sequence of cycle-level actions on bus signals.

In this paper, we show how to integrate individual TLM models written in SystemC into the open-source QEMU and commercial Wind River Simics full-system simulators. These simulation platforms come from a different market and different user base than SystemC with its EDA heritage. They have been designed for high-speed simulation of target software rather than for the detailed design of hardware. Their native modeling style is TLM. Some of the functionality and semantics of SystemC do not fit immediately into the fast full-system TLM simulation style, and we need some intelligence in the integration to handle the "impedance mismatch" while still enabling high-speed simulation.

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 exactly the same simulated system state. For VPs, 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.

Checkpointing is becoming more important as simulated systems increase in complexity and workload size and see more use for software development and test. 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. Needless to say, in these cases checkpointing is necessary to avoid repeating this bring up [1]. Checkpoints can also be used to transport bugs for bug reporting between geographically or organizationally distributed teams [2].

Section II introduces previous work in the field of SystemC and VPs and the ability of checkpointing the simulation; in Section III we introduce work done with SystemC and QEMU, and how was applied to Wind River Simics. In Section IV we detail how we add checkpoint features to the SystemC kernel. The tests and experiments are presented in Section V. Finally, the conclusions of this paper are presented in Section VI

II. Simulation Systems

There are many products relevant to the virtual platform industry, notably: Wind River Simics [3], Imperas' Open Virtual Platform [4], Synopsys' Innovator family (which now includes the CoWare and VaST simulation frameworks) [5], QEMU [6], Bochs [7], IBM's Mambo and CECsim tools, and ARM's fastsim models. Virtual platform tools are designed to help develop hardware and software products, and have feature sets geared towards developers. Their defining quality is that they model a particular target hardware, independent of the host machine they are run on.

A. QEMU

QEMU is an open-source generic machine emulator based on dynamic translation. Currently QEMU emulates the following processors: 32 and 64 bit x86 PCs, ARM, SPARC, MIPS, and Coldfire, with several other platforms in various stages of development.

The communication between a CPU emulator and the emulated devices is done via registered callback functions for each memory region of the system bus. Then, when the CPU emulator

Manuscript received September 23, 2010; revised March 25, 2011; accepted December 07, 2011. Date of publication January 27, 2012; date of current version December 19, 2012.

M. Montón is with the CAIAC, Catlunya 08193, Spain (e-mail: marius.monton@uab.cat).

J. Engblom and M. Burton are with the Wind River, Stockholm 16483, Sweden.

Digital Object Identifier 10.1109/TVLSI.2011.2181881

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. QEMU devices respond immediately to devices accesses, and do not account for any processing time in the device or any bus occupancy.

QEMU is a monolithic simulator, where each simulation setup is compiled into the binary, and with no support for dynamic reconfiguration at run time.

B. Simics

While QEMU provides an interesting VP example, it is not unique, and to prove that our methodology is more widely applicable, we have also integrated SystemC with Wind River Simics.

Like in QEMU, Simics assumes that all device models return immediately from a transaction call. Using special memory map objects, Simics can map any device in the system into any location(s) in memory.

Simics device models (and any other simulator module) can be written in any language, as long as the result of the compilation is a DLL or shared object file that adheres to the Simics API [8]. Unlike QEMU and SystemC, Simics is not statically linked but modules are loaded as needed at runtime. The system configuration can also be changed during a simulation run, unlike SystemC and QEMU.

C. SystemC

SystemC is a different type of simulation system from QEMU and Simics. It is a general simulation kernel designed for models from cycle-accurate and pin-accurate modeling of hardware behavior all the way up to abstract bandwidth and behavior models. Unlike QEMU and Simics, SystemC allows for device models to contain active threads $(\mathrm{SC\_THREAD})$ as well as purely event-driven objects $(\mathrm{SC\_METHOD})$. This is very similar to the process concept of VHDL. SystemC is based on C++, and all objects in the SystemC simulation are C++ objects.

SystemC is an event-based simulator with potentially multiple active simulation contexts. When simulation begins, the kernel finds the top event on its event queue and executes or resumes the thread or method sensible to that event.

Using the $\mathrm{wait}()$ call, it is possible for a thread to suspend its execution for a certain amount of time. This means that calls from one system model to another might block and not return until some amount of simulation time has passed. It also requires the use of a user-level threading system to store the execution state of the model, including local variables on the stack.

D. SystemC TLM-2.0

The implementation of TLM within SystemC has been standardized by OSCI SystemC TLM-2.0 [9]. SystemC TLM-2.0 supports models which use both threads and methods. It uses direct function calls for communicating transactions between simulation models. It defines two main programming styles for models: LT and AT, as discussed below. The design of TLM-2.0 aims at allowing for the modeling all buses and bridges in a system. It is up to the modeler to build the components that map from addresses to devices (which is provided by the platform in QEMU and Simics).

The use of $\mathrm{SC_THREAD}$s with TLM modeling is not recommended. Rather, the literature on TLM modeling recommends relying on $\mathrm{SC_METHOD}$s to reduce number of events posted and the kernel overhead, and thus improve simulation speed [10], [11].

1) Loosely Timed (LT): This style uses a blocking interface, allowing up to two timing points for each transaction: the call to the blocking call and the return from it. These two points can occur at the same simulation time or at different times (if the target of the call was an $\mathrm{SC\_THREAD}$ that called $wait()$). Alternatively, the call can return immediately but add an annotation that indicates that the caller has to account for the time elapsed. The loosely timed coding style supports the modeling of timers and interrupts, sufficient to boot an operating system and run arbitrary code on the target machine.

LT is fairly similar to the execution models of QEMU and Simics, as long as the time for a call is kept to zero. Annotating time or using $\mathrm{wait}()$ complicates the integration.

2) Approximately Timed (AT): This codying style uses a nonblocking interface and allows any number of timing points per transaction. Each timing point is marked by a complete call from one model to another. Even as the function call returns, however, the transaction is not necessarily complete. Rather, the result of a transaction from a processor to a device is typically reported by a later callback from the device to the processor.

This back-and-forth communication style is not well-suited for high-performance simulation since the processor simulator will have to either model complex read queues or just stall and wait until the result comes back. QEMU and Simics do not normally support this type of communication between processors and devices.

The most important characteristic of this style is that it supports the integration of TLM models with RTL or cycle-level models. It lets designers model the timing of buses, including effects of bandwidth limitations, bus latencies, and bus arbitration, which is not visible at the LT level.

III. Simulator Integration

When integrating two simulation subsystems from two different simulation environments, the key question is which environment to use as the master. One simulation will need to be the master, deciding how and when to drive the simulation and the simulated time forward, and the other will have to be a slave or subsystem to the master simulation.

In two of the integration examples we present here, we have used SystemC as the slave to QEMU and Simics. This solution makes sense when we expect most of the work in the complete simulation to take place in QEMU or Simics, and the SystemC subsystem to contain only a small part (a few peripheral devices, typically) of the complete simulation.

On the other hand, when most of the system exists as a SystemC system, and we want to reuse only a few components from QEMU, the QBox solution makes the most sense where SystemC is the simulation master. Simics also supports being embedded into other simulation systems, as a standard Simics feature not related to the work presented in this paper.

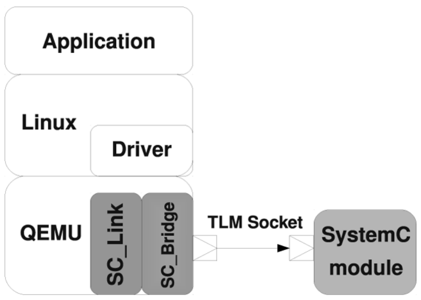

Fig. 1. QEMU-SC block diagram.

A. *QEMU-SystemC*

In this integration, we use device models written in SystemC as emulated devices in QEMU. The goal is to make as few changes to QEMU as possible. This solution uses QEMU as the simulation master, and the SystemC devices as the simulation slave. To achieve this, we introduce a link between the two worlds. This a QEMU device that gets called from QEMU like any other QEMU device, and then calls into the SystemC subsystem to get the simulation work done. The original name for this project was QEMU-SC.

1) Architecture: The first step is to have a QEMU proxy device. This device is called sc_link. It handles the memory mapping of the device towards the QEMU system, and receives data reads and writes from QEMU.

These accesses are passed to the SystemC device, and the responses from the SystemC device are sent back to QEMU. This step is done through a *SystemC bridge* (or sc_bridge), that also manages synchronization of time between the simulators.

The SystemC bridge will act as a SystemC TLM *initiator* for the transactions, with the actual SystemC device as the *target*. When called, the bridge creates and fills in the TLM transaction and performs the transaction operation, managing SystemC simulation time to finish the transaction. The block diagram is shown in Fig. 1, with SystemC blocks in dark gray.

With this architecture, adding SystemC support to another QEMU virtual platform target is as simple as writing a new sc_link module for that specific platform (and its bus fabric).

The SystemC bridge, the SystemC device, and the OSCI simulator are all compiled and linked with the rest of QEMU to obtain the QEMU executable for the final complete system.

Communication between the bridge and the SystemC device is done through a TLM socket. In our implementation, we use the GreenSocs Generic Protocol Socket [12]. This code base 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".

We use the LT interface of the SystemC TLM-2.0 models, as that ensures that the function called returns with all response data ready to use. Since TLM-2.0 mandates that all models expose both an LT and an AT interface, this should work with all TLM-2.0-compliant models.

The TLM 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, and the data to write in case of write access. The rest of the transaction is static throughout the simulation, to reduce the amount of memory allocation performed.

If a bus is capable of identifying pure memory transactions (such as a PCI bus), the SystemC link tries to use direct memory interface (DMI) functions when the CPU accesses a memory region of the device. We implemented this mechanism for the case of the x86-based platform and PCI devices registering themselves as a memory devices.

2) 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 be unusably slow.

Our solution to time synchronization is based on a lazy connection, and the notion of "quantum time" introduced in SystemC TLM-2.0. We synchronize the two simulators only when necessary, and not on every cycle executed in either simulator.

In the simplest case, we assume that SystemC is used to model a small number of devices of the VP. 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 only activated when the subsystem is accessed. However, this is not sufficient.

Each time a SystemC device is accessed, it might generate and post events for immediate or future action. To take this into account, every time a transaction is sent from QEMU into the SystemC subsystem, the SystemC bridge needs to check if there are any new events posted on the SystemC kernel queues.

If a future event exists, the corresponding event time is posted in a QEMU event queue. When QEMU reaches that time, a callback is called and the SystemC bridge is invoked. The SystemC bridge starts the SystemC simulation to again, running simulation so that it catches up to the current time in QEMU. This will allow it to process the event that had been scheduled. When this operation is complete, the bridge checks again to see if there are new outstanding SystemC events, and if so once again posts a future event in QEMU.

Unfortunately, there is no "standard" interface to the SystemC kernel to provide the required information. To be able to ask to OSCI kernel for any of its state we had to make a small modification to SystemC kernel. We made our bridge class as C++ "*friend*" class of the $sc\_simcontext$ in the SystemC kernel. In this way, our bridge can "peek" at the events list inside the simulator.

This only deals with events posted in virtual time inside of SystemC. This should cover all modeled hardware activity, but it does not cover the case that SystemC models interact directly with the user or host computer. For example, a SystemC model that communicates with an external Xterm in order to implement a simulated serial console will have an OS-level thread active waiting for asynchronous input from the outside world. In this case, the SystemC system has to be called in response to outside events, and not just its internal events. One way to solve this is to use *temporal decoupling* concept of TLM-2.0, and make sure that the SystemC subsystem is called at each "global quantum", essentially making the device model poll for new asynchronous events.

The mechanism responsible for connecting the "outside world" to the SystemC model must "notify" the SystemC bridge (using an inbound transaction) 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.

In the case of devices that have external I/O, because the device is "frozen" most of the time, handling events only at the pace given by the global time quantum might result is poor responsiveness to asynchronous inputs from outside the simulation (depending on the length of the quantum and the overall speed of simulation). The better solution is to move the interaction with the outside world from SystemC into QEMU, and have the QEMU module notify the SystemC subsystem with a transaction when an event occurs. This is the solution mandated in the Simics integration described as follows.

In the case that devices do not interact with the host, the quantum-based synchronization only updates the SystemC time to be the same as QEMU time, which is a very fast operation.

3) Changes to SystemC Main: As we put the entire SystemC subsystem as a slave of QEMU, the $sc\_\mathrm{main}()$ function is removed from the simulation. The simulation $\mathrm{main}()$ function belongs in the QEMU basic code, and the simulation starts from QEMU and calls SystemC as a subsystem. The current OSCI kernel implementation calls callback functions $(\mathrm{before\_end/end\_of\_elaboration},\ \mathrm{start\_of\_simulation})$ when $sc\_\mathrm{start}()$ is invoked, so no negative effect is observed when we removed $sc\_\mathrm{main}()$.

B. Wind River Simics-SystemC

We also implemented an integration between Simics and SystemC. Just as with the QEMU case, this work involves managing the OSCI kernel, making the SystemC kernel a slave of the Simics simulation. We did this work based on Simics 4.0, and a later version has been shipping as a standard feature of Simics since Simics 4.2.

1) Architecture: Overall, the approach taken with Simics is very similar to the approach taken with QEMU. There are some differences in details though, which we will describe here.

Thanks to the services provided by the Simics platform and its memory map system, we did not need a QEMU sc_link equivalent. Simics devices only receive transactions that are relevant to them (as setup in the system memory maps), regardless of the system bus of the emulated platform. Virtual memory translation and PCI configuration are handled for us. Thus, we only need to use the SystemC Bridge module for the integration.

The Simics SystemC bridge deals with time synchronization, and contains the transaction translators necessary to map Simics operations into SystemC transactions. It also contains the OSCI kernel, just like in the QEMU case. A notable difference is that Simics modules are dynamically loaded modules, but including the OSCI kernel inside a such a module turned out to be no problem at all.

The Simics side of the bridge can use all mechanisms and functions from Simics API, and it is responsible for interfacing all I/O between SystemC modules Simics. It looks like any other Simics device or module from the perspective of the rest of the Simics system.

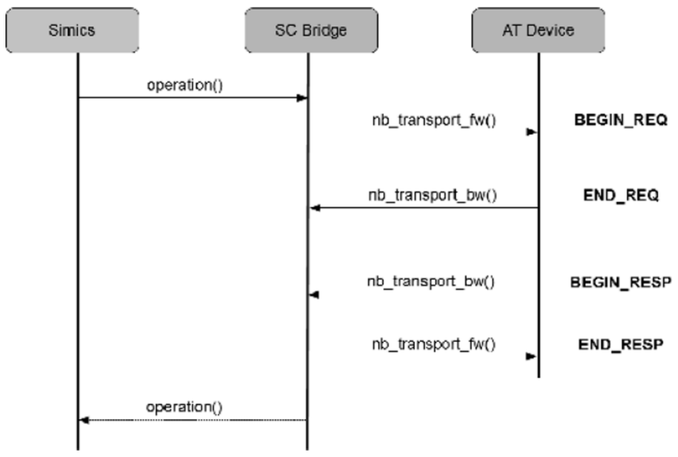

Fig. 2. Communication schema between Simics and an at device.

2) Implementation: The bridge between two simulators is in charge of two related tasks: adapt different transaction and data moving between both simulators; and manage simulation time between the two simulators.

In Simics, just like in QEMU, 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 mechanisms fits perfectly with OSCI TLM 2.0 blocking calls in untimed (UT) or loosely timed (LT) calling style. Thus, our Simics bridge works with this type of TLM communication. For pure event-driven models (models that use events to model time such as the delay between an operation start and its completion interrupt), the timing and semantics are the same as when the device is run in a SystemC environment.

Unlike for QEMU, we also investigated directly connecting from Simics to an AT-style TLM-2.0 model. In this case, although the communication between the Bridge and the TLM model is using non-blocking calls, the bridge finishes all the phases of the communication before returning to Simics, as shown in Fig. 2. To account for the time spent, we then stall the processor in Simics (meaning that it does no work until the time when the AT transaction returns). This way of using an AT model does not allow pipelined operations and bus transactions, but it does allow accurate timing to be computed for each operation in isolation.

The temporal decoupling mechanism is also supported by this bridge, the amount of time (quantum time) that the SystemC device can run ahead of the current Simics simulation time is configurable by the user during the configuration phase. In later generations of the bridge, this was removed as it is strictly unnecessary for well-behaved devices that do not interact directly with the outside world.

The bridge can manage more than one SystemC device, mapping into different memory regions and managing transactions accordingly.

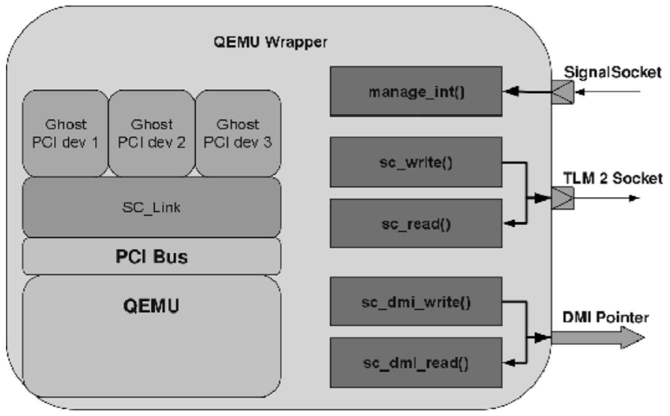

Fig. 3. Wrapped QEMU block diagram.

C. QBOX

Another way to use the QEMU simulator with SystemC is to make QEMU itself into a SystemC module, and let SystemC be the master of the simulation. Typically, QEMU processor models would be acting as initiator and QEMU devices as targets in the SystemC system. It makes QEMU a slave of the SystemC master, inverting the relationship.

To achieve this, 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()) (as depicted in Fig. 3).

1) Architecture: QEMU is based on a endless simulation loop. To facilitate the integration as a slave simulator, this loop is extended by a call to a synchronization function that computes the difference between QEMU time and SystemC time. If QEMU time is far ahead of SystemC time, QEMU yields.

Our SystemC wrapper contains an $\mathrm{SC\_THREAD}$ that runs the QEMU modified main loop and yields to the SystemC kernel when necessary, using $\mathrm{wait}()$. Also, the wrapper has two functions ($sc\_\mathrm{read}/sc\_\mathrm{write}$) to allow QEMU to use them to access to SystemC devices when requested. These $sc\_\mathrm{read}/sc\_\mathrm{write}$ use a TLM-2.0 socket in the same way as has been explained in section 3.2.3 (Implementation).

The wrapper works together with a special version of the sc_link that connects itself to the QEMU system bus, providing access to the rest of the SystemC bus, and 'ghosting' the other SystemC devices so that the QEMU system knows they are there. The sc_link device captures all data accessed from QEMU to those ghost.

2) Implementation: We wrote a TLM 2.0 Initiator module named qemuWPSC that wraps QEMU and becomes the SystemC QBox module.

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

The wrapper is designed to use Temporal Decoupling, so the synchronize function uses the TLM 2.0 Quantum Keeper to manage its own simulation time and only yield to the SystemC simulator when QEMU time is ahead of SystemC time by more than the global quantum.

When the QEMU wrapper yields to SystemC, it freezes the QEMU simulation before passing to SystemC simulator. Once execution comes back to the wrapper, QEMU execution is resumed again.

When QEMU accesses the ghost device plugged in the system bus, it calls a callback function in sc_link. This function captures information regarding the access type (read or write, exclusive access, etc.), address, and data (in case of a writing). This data is then passed to the wrapper who builds the corresponding TLM transaction. Once built, the transaction is send through the TLM Socket to the SystemC device using the blocking transaction mechanism (loosely timed).

As this mechanism allows Targets to yield to the simulator (calling $\mathrm{wait}()$), the wrapper suspends QEMU execution before the data communication begins. In this manner, we ensure that QEMU is stopped in case a target yields to Simulator.

Once the blocking transport returned, the wrapper breaks the transaction up, extracting the system bus data and sends it back via sc_link, and resumes QEMU execution again.

Because bus accesses in QEMU are synchronous, QEMU expects a bus access to complete once the callback function returns. For "AT" style slaves, this is handled in as previously described in the previous section and depicted in Fig. 2.

IV. CHECKPOINTING SYSTEMC

To support checkpointing in SystemC (and indeed in any simulator system), there are the following 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 and the current simulation time.
- Saving and restoring the simulation configuration in terms of which simulation models form part of a VP, and how they are connected.

We have addressed these problems for SystemC models, using the OSCI 2.2.0 SystemC kernel, when used within the context of the Simics SystemC bridge as described above. Simics provides a reliable and proven infrastructure for checkpointing and its related file handling [8], letting us focus on the issues of storing and restoring the state of a SystemC simulation. Note that QEMU has a snapshot system that should also support checkpointing of SystemC subsystems in a similar way, but we have not explored that option.

Current SystemC implementations do not support checkpointing. Because of the threaded semantics and compiled C++ basis of SystemC, adding it to anything but a subset of SystemC seems infeasible. Our approach is to inspect the SystemC kernel to access to its state variables; provide a mechanism to designers to explicitly declare checkpointable state variables; and add checkpoint features to SystemC primitive channels (which modelers use as-is).

A. Model State

Checkpointing requires a model to explicitly define the state that will be saved and restored. In our implementation, we have used the GreenControl [13] 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 a special $gs\_\mathrm{param}<>$ template.

From the 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. The GreenControl system allows access to parameters from outside the module that declares them, even when the simulation is not running, providing the back-door access we need to get and set the simulation state of a model.

The main limitation of our system is that only module data members using the $gs\_\mathrm{param}\ <>$ mechanism are checkpointed. SystemC ports are currently not covered. This means that port values are not saved, and SystemC modules have to be written with this in mind. For TLM models, this should be a non-issue, since communication does not use ports but rather function calls.

It also means that we cannot checkpoint SC_THREADs in middle of their execution. The problem is that the SystemC kernel maintains a separate stack for each thread, and this stack contains the current execution state of the thread (as a saved program counter value in the middle of some function) as well any local objects and variables. Saving and restoring this is not possible in a portable way, since each compilation of a program will likely result in a very different stack layout and code layout. All C++ serialization libraries also preclude saving thread state, allowing only object state to be saved, which is perfectly analogous to our approach.

Another small limitation is that we currently assume the SystemC configuration to be a fixed subsystem, represented as a single Simics object. The names of checkpointed parameters and the target of stored events depend on the SystemC module hierarchy, and thus changing the hierarchy will break the restoration of old checkpoints. It is not a big problem in practice, since the point of restoring a checkpoint is to get back to a previous state.

B. SystemC Kernel

We have to save and restore the information that the kernel needs to continue a simulation. This includes the actual simulation time and the event queues, primitive channel data, etc.

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, and pending events are re-posted to the event queues, and primitive channel data is restored.

1) Checkpointing Events: We access the event queue through the OSCI SystemC kernel class $sc\_\mathrm{simcontext}$. This class encapsulates the SystemC kernel and manages the simulation. The $sc\_\mathrm{simcontext}$ class manages four different event queues, two for events involving methods, and two for thread processes. These queues store the pending event list for dynamic sensitivity and static sensitivity of the methods. We add a single method to the queue mechanism in order to get all pending events without destroying the list (existing methods in the kernel had the unfortunate side-effect of emptying the queues, which is not acceptable).

When resuming from a checkpoint, the events for the queues are posted back again. Once we have posted all events, the simulation state is the same as it was when the checkpoint was taken.

2) Checkpointing Primitive Channels: SystemC provides channels as a mechanism to communicate between modules. Channels hold simulation state, since values can be stored in them at one point in time, and retrieved at a later point in time. In order to checkpoint these channels, we need to add some backdoor methods to access their state.

The implementation for the $sc\_\mathrm{fifo}<T>$ channel is shown as follows.

3) $sc\_\mathrm{fifo}<T>$: sc_fifo is a SystemC predefined primitive channel modeling the behavior of a FIFO buffer. The $sc\_\mathrm{fifo}<T>$ class is designed as a C++ template, meaning that can manage any data-type using the same methods.

To checkpoint it, we added the following two new methods to the class: restore() and checkpoint().

- std::string $sc\_\mathrm{fifo}<T>$::checkpoint() returns a string with all relevant data for the sc_fifo: number of slots and data stored in it.
- bool $sc\_\mathrm{fifo}<T>$::restore(std::string $\mathrm{chkp\_data}$) receives a string with all checkpoint data and restores the sc_fifo buffer to the given serialized state.

These two methods will work if data-type T supports converting to meaningful string. If the user needs to use a special data-type that does not support stringification, the methods can be overridden to provide a special checkpointing method.

It is important to emphasize that these new checkpointing methods do not disturb normal simulation of the system, nor do they affect the simulation speed or the original system behavior. They are used only in the checkpoint or restore phase of the simulation.

4) FSMs: Finite-state machines are usually described in SystemC using SC_THREADs. Rewriting such threads into methods is easy, it is just a matter of using a checkpointed variable inside the model to hold the current state, and using events to drive any recurring activity. Note that care has to be taken when external functions are called from the FSM, to ensure that they do not call SystemC wait(). Essentially, the model is turned from an active thread into an event driven passive model, which is a common modeling style also.

C. Checkpoint Manager

In order to enable checkpointing in the SystemC kernel, we introduced a new class which manages all operations needed to perform the restore or checkpoint correctly. This class is a singleton, since only one is needed per simulation instance. The following methods are offered by the checkpoint manager.

- std::string checkpoint() takes a checkpoint at current simulation point. This methods return a string with all checkpointing data. This string is managed by Simics as the checkpoint state of the SystemC subsystem, but Simics does not need to interpret it in any way.

- bool restore(std::string) restores saved events and data into the simulation kernel. The method receives the string previously created by a checkpoint() call in another simulation session.

D. Related Work

SimOS and IBM Mambo full-system simulator [14], [15] use a checkpointing mechanism very similar to that which we propose, with model state explicitly separated from the simulation kernel state. Typical uses for checkpoints in Mambo are to save complex system setups and to switch between different implementations and different abstraction levels for the same models.

The Boost C++ serialization library can save and restore the state of C++ objects in a program [16]. However, using this library with SystemC would have required a rewrite of the SystemC kernel to use serializable objects to store all state, and it is not clear how this would work with the cooperative multitasking nature of the SystemC kernel. 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 the solution used by Synopsys [17] and Cadence for SystemC. The memory-dump solution has several limitations compared to our approach. Since memory dump 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, nor can it be used with an updated or different model code (even a minor change such as a Linux kernel version or different set of system libraries is likely to break it).

V. Results

A. SystemC and Virtual Platforms

1) QEMU-SystemC Bridge: We build different test-benches: all tests were done using an unmodified Debian GNU/Linux running as a guest on QEMU. We used two of the QEMU platforms: intel x86 PC and the ARM VersatilePB [18].

We started with a simple SystemC device that publishes a register file. This register file is accessed through the socket. We mapped this register file directly to the AMBA bus in the ARM platform and to a single memory BAR for the x86 PC, via PCI. For both platforms, we wrote a Linux driver that presented the SystemC device as a character device to the system, allowing user applications to read and write to the device as a file system and the driver sends the operation to the VP system bus.

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 period of time (we set this time to 10 s). The interrupt generation can be disabled with a read of 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 (re-)posts the event again.

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 sufficient.

Our third experiment was designed to test SystemC TLM-2.0 DMI access. This functionality is transparent to the OS, driver, and user application. This enhancement should give better performance when accessing memory devices than simply using standard transactions across the Sockets but 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 of using bus bursts.

2) Wind River Simics Bridge: We tested the Simics SystemC bridge using a SystemC model of an NS16550 [19] serial port (UART). The UART device was written from scratch in TLM 2.0 LT/AT style, with no temporal decoupling support. The system used was an AMCC "ebony" board [20], based on a PowerPC 440GP SoC. This board was running a full Linux kernel, as well as a synthetic test for the UART device.

While running Linux, we changed the configuration of the system to use the SystemC UART instead of the native Simics UART. The performance impact of using the fairly complex SystemC model compared to the Simics standard NS16550 model was imperceptible in normal use.

When using a micro-benchmark program that pushed characters to the UART as quickly as possible, the performance was reduced by about 5% when using the SystemC UART instead the Simics standard UART model. This 5% penalty is probably caused by the data translation between Simics and SystemC and the overhead of synchronization, concluding that the SystemC bridge does not present a significant performance problem.

B. QBOX

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.

To demonstrate the power of this configuration, we built a more complex system involving two Ethernet controllers modeled in SystemC and a QEMU x86 PC model. The Ethernet controller modeled was the ne2000 PC card by National Semiconductor, based on an DP8390 Ethernet chip [21]. The SystemC module was written from scratch based on the ne2000 device model already present in QEMU. The model has a TLM-2.0 socket to interface to QEMU and one Initiator and one Target port to connect to the Ethernet side. Also, it has another socket (signal socket) to manage the interrupt interface to the CPU.

These two devices are connected to the QEMU wrapper using a Router (GreenRouter) from GreenSocs [22]. The router has the same memory map used by the sc_link, and routes the transactions to the correct device accordingly to that memory map.

TABLE I
PERFORMANCE TEST RESULT FOR gs_param

Data Type	Runtime (sec.)	Relative runtime
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

TABLE II
TIME RESULTS FOR FSM DESCRIPTION USING sc_thread AND sc_method. (RUNTIME IN SECONDS)

Test	SC_THREAD	SC_METHOD	SC_METHOD & gs_param<>
1	115,053	85.858	109.678
2	130.67	119.639	195.217
3	-	86.044	318.567

Tests show a penalty of about a 12% compared to the same system described in QEMU using QEMU's normal device models. This penalty is due to the synchronzation mechanism and the adaptation of the data for TLM transactions.

C. Checkpointing

1) gs_param *Overhead:* In our SystemC implementation of checkpointing, there is a potential additional performance impact from the use of the GreenSocs $gs\_\mathrm{param}<>$ mechanism. We created a performance test [23] (cf. tbl. 1) consisting of two TLM-2 devices: A master exercises 100 million TLM-2 write transactions to a slave which writes the received payload to an unsigned int, a sc_unit or gs_param variables. This means an intensive usage of parameters inside the tested module.

Table I shows that accessing model variables which have been instrumented with gs_param to be quite efficient. The results show a penalty of about 2%—depending on the data type—as compared to the same model without internal variables modeled with gs_param.

2) SC_THREAD *to* SC_METHOD *Overhead:* We evaluated the simulation speed for different FSMs described using both SC_THREAD and event-driven checkpoint-friendly SC_METHOD. The translation was done by hand, first checking the correctness of the translation and later measuring simulation time for the equivalent description. In these tests we also include the description of the same FSM using $gs\_\mathrm{param}<>$ 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.

We summarize the following results in Table II.

- Test #1 is a simple 3-state FSM with only 1-bit input signal and 1-bit output signal.
- Test #2 is an iterative multiplier based on successive additions.
- Test #3 is a Fast Discrete Cosine Transformation module designed for JPEG compression. This module is designed to receive the data to be processed in a single transaction; then the module performs the calculations and send the result back to the VP. 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 five intensive used variables in $gs\_\mathrm{param}<>$.

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

void systemc_chckpnt_test::function()
{
  my_event.notify(1, SC_US);
}
```

Fig. 4. Test code example.

3) 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 Fig. 4. This device exhibits all the 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. When the register is written, a periodic event is triggered at each fixed time (1 us).

Our test consists of: start the simulation and do a write to the register to start the periodic event. Next, take a checkpoint and quit Simics. Start Simics again and we resume the simulation from the checkpoint. We then check 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, and the simulation can be restored using our strategy.

4) Validating Model Updates: Model updates often change the code for a specific part of a model. The aim is to allow them to re-use the same checkpoint data, as long as the new set of model data is a superset of the old data such as adding a new register. We created a simple SystemC device model containing a single memory-mapped register. We then executed a 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 start). Thus, we show that we can update models and use old checkpoints.

Another side of "updatability" is that it 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 can be tested, or when a bug is found that can easily be bypassed by changing a single value (or a set of values).

5) Checkpoint Size: Since the checkpoint system that we use here only stores the essential data for a simulated system, it will generate very compact checkpoints in general. For our running example, the checkpoint was about 88 kb in size—most of

```
OBJECT tgc0 TYPE timer_chckpnt {
    ...
    systemc_time: 0x5c10c637307
    gs_all_param_value:
      "timer_chckpnt.interrupt_status=0;
      timer_chckpnt.register_bell=1;
      timer_chckpnt.register_control=1;
      timer_chckpnt.register_countdown=100000;
      timer_chckpnt.register_status=0;"
    methods_events_list:
       "6326694671111\n6426693333333;
    timer_chckpnt.timer_manager;"
}
```

Fig. 5. Checkpoint information for the SystemC timer device in Simics.

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 Fig. 5.

That can be compared to the overall process size of the simulation, at 263 688 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.

VI. Conclusion

We presented our work on adding checkpointing to SystemC, in conjunction with a simulation framework such as Windriver Simics. This support is reduced to a subset of the SystemC specification. However this subset is complete enough to simulate any TLM model for Virtual Platforms.

We added checkpoint support directly in the OSCI kernel, allowing any tool or framework to use it easily.

We also presented our work and mechanisms to join two different simulators like the SystemC kernel and the Virtual Platform simulation kernel. This work has been done for two different Virtual Platforms resulting in a full checkpointable and SystemC capable Virtual Platform in the two cases.

References

[1] M. Bergqvist, J. Engblom, M. Patel, and L. Lundegård, "Some experience from the development of a simulator for a telecom cluster (CPPEMU)," presented at the 10th IASTED Conf. SW Eng. Appl., Dallas, TX, 2006.

[2] J. Engblom, "Transporting bugs with checkpoints," presented at the Syst., Softw., SoC, Silicon Debug Conf. (S4D), Southampton, U.K., 2010.

[3] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A full system simulation platform," *Computer*, vol. 35, pp. 50–58, 2002.

[4] Imperas, U.K., "Imperas Webpage," 2010. [Online]. Available: http://www.ovpworld.org

[5] Synopsys, San Jose, CA, "Innovator Webpage," 2009. [Online]. Available: http://www.synopsys.com/tools/sld/virtualprototyping/pages/innovator.aspx

[6] F. Bellard, "QEMU, a fast and portable dynamic translator," in *Proc. Annu. Conf. USENIX Annu. Tech. Conf. (ATEC)*, 2005, pp. 41–41.

[7] Bochs, "Bochs Webpage," 2010. [Online]. Available: http://bochs.sourceforge.net/

[8] J. Engblom, D. Aarno, and B. Werner, *Full-System simulation from embedded to high-performance systems*. New York: Springer, 2010.

[9] O. T. Wg, "Requirements Specification for TLM 2.0, Version 1.1," 2007.

[10] M. Schnieringer and K. Brand, "SystemC: Key modeling concepts besides TLM to boost your simulation performance," 2010.

[11] S. Swan, "An introduction to system level modeling in SystemC," 2.0 2001.

[12] GreenSocs, U.K., "Greensocket Website," 2009. [Online]. Available: http://www.greensocs.com/en/Projects/GreenSocket

[13] GreenSocs, U.K., "Greencontrol Website," 2009. [Online]. Available: http://www.greensocs.com/en/Projects/GreenControl

[14] J. L. Peterson, P. J. Bohrer, L. Chen, E. N. Elnozahy, A. Gheith, R. H. Jewell, M. D. Kistler, T. R. Maeurer, S. A. Malone, D. B. Murrell, N. Needel, K. Rajamani, M. A. Rinaldi, R. O. Simpson, K. Sudeep, and L. Zhang, "Application of full-system simulation in exploratory system design and development," *IBM J. Res. Dev.*, vol. 50, no. 2/3, pp. 321–332, 2006.

[15] M. Rosenblum and M. Varadarajan, "SimOS: A fast operating system simulation environment," Stanford, CA, 1994.

[16] Boost, "Boost Serialization V.1.36," 2010. [Online]. Available: http://www.boost.org

[17] S. Kraemer, R. Leupers, D. Petras, and T. Philipp, "A checkpoint/restore framework for SystemC-based virtual platforms," in *Proc. Int. Symp. Syst. Chip (SoC)*, 2009, pp. 161–167.

[18] Arm, U.K., "Arm Versatile Website," 2010. [Online]. Available: http://www.arm.com/products/DevTools/VersatileFamily.html

[19] N. Semiconductor, PC16550D Universal Asyncrhonous Receiver/Trasnmitter Wifh FIFOs 1995.

[20] Virtutech, Sweden, "Simics virtual platform for the AMCC ebony (PPC440GP)," 2010. [Online]. Available: http://www.virtutech.com/solutions/virtual_platform/powerpc/amcc/ebony.html

[21] N. Semiconductor, "DP8390D—NIC network interface controller," 2010..

[22] GreenSocs, U.K., "Greenrouter Website," 2009. [Online]. Available: http://www.greensocs.com/en/Projects/GreenParts/GreenRouter

[23] C. Schröder, W. Klingauf, R. Günzel, M. Burton, and E. Roesler, "Configuration and control of SystemC models using TLM middleware," in *Proc. 7th IEEE/ACM Int. Conf. HW/SW Codesign Syst. Synth. (CODES+ISSS)*, 2009, pp. 81–88.

Màrius Montón was born in Barcelona. He received the Engineering degree in computer science and the M.S. degree in microelectronics from the Universitat Autònoma de Barcelona (UAB), Barcelona, Spain, in 2003 and 2006, respectively.

In 2000, he joined CEPHIS technology transfer node from the regional IT network as an R&D engineer. Since 2004, he has been a part-time teacher with the Departament de Microelectrònica: i Sistemes Electrònics, UAB. In 2007, he joined GreenSocs, working on joining SystemC and TLM-2.0 with other simulators, developing projects for Virtutech and Intel.

Jakob Engblom received the M.Sc. degree in computer science and the Ph.D. degree in real-time systems from Uppsala University, Uppsala, Sweden.

He is Technical Marketing Manager for the Simics product line at Wind River, Stockholm, Sweden. He has written and presented over 100 articles and talks on a variety of embedded systems topics since 1997. Since 2002, he has been working with the Simics full-system simulator, first at Virtutech and then at Wind River. His interests include debugging, computer simulation, virtual platforms, embedded software development, and computer history.

Mark Burton received a degree in computer systems engineering from Warwick University, Coventry, U.K. He completed the Ph.D. degree in artificial intelligence within education, specifically focusing on the simulation of high level collaborative learning processes.

He is the founder of GreenSocs. He then worked for ARM becoming the manager of the modeling group. At this time he was also the chair of the OSCI TLM WG. Recently Mark formed GreenSocs with a number of aims, notably to support the entire Electronic System Level industry to better inter-operate, but also to support research and development environments to have more contact with relevant industrial tools and techniques. He is currently the chair of the OCP-IP SLD WG.