Checkpoint and Restore for SystemC Models

*Màrius Montón**
GreenSocs
marius.monton@greensocs.com

Jakob Engblom
Virtutech
jakob@virtutech.com

Mark Burton
GreenSocs
mark.burton@greensocs.com

Abstract

We present preliminary work in the field of saving and restoring model state within a SystemC simulation environment. Save and Restore (or checkpointing) is a useful technique that can greatly assist target software and simulation model development and debug. In contrast to other approaches that aim at saving and restoring the state of an entire simulation process, we investigate mechanisms by which only the essential simulation state is saved. This makes the checkpoints far more compact, and saved simulation states can be moved between host machines, and be used with updated or completely different simulation models.

Our results indicate that SystemC models written to certain coding guidelines can be saved and restored reliably. As a result, virtual platforms and platform components written in SystemC can be made more useful to software developers, and support smarter workflows.

1. Introduction

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

Checkpointing is a key workflow enabler for systems and software development using a virtual platform. With checkpointing, a software developer can save and restore their work at any point and resume it later without having to keep the simulator running. A repetitive simulation procedure such as booting an operating system and loading a set of software applications onto the system can be done once, saved, and then used may times, saving time and ensuring multiple developers have identical system setups. This is getting more and more important as simulated systems increase in complexity and workload size. We have seen cases where a system bringup takes hours, as it involves the simulation of many billions of instructions across hundreds of processors, including reboots and software. Needless to say, in these cases checkpointing is necessary to avoid repeating this [1]. Checkpointing also makes it possible to store a library of booted and configured systems of various forms, for use in regression testing or to try various alternative microarchitectures on the same booted software load.

To be truly useful in a software development context, a checkpoint has to be portable across hosts and simulator versions. As checkpoints are exchanged between different user groups and different companies, it is impossible to know where they might end up. As an example, it must be possible to use the same checkpoint on a 32-bit x86 Linux host and on an UltraSPARC Solaris host. It must also be possible open a checkpoint taken in an earlier version of a virtual platform in a later version.

During simulator development, checkpoints make it easy for customers of a modeling service to report bugs and test the fixes provided. By using a checkpoint of a situation where the model fails to execute correctly, it is very easy for the modeling service to reproduce and locate an error.

In a similar vein, checkpoints are useful during iterative model development where most parts of a model are marked as "unimplemented". As soon as software actually accesses unimplemented features, a checkpoint is taken, the model is updated and simulation restarted from the checkpoint to see that the software reacts correctly to the now implemented register or feature. This enables very rapid development of virtual platforms, especially for existing hardware systems [1][2].

Checkpointing was originally developed in the mid-1990s to support changing the level of abstraction in a simulation model, from a fast approximative (CPU) model to a detailed microarchitecture model [3, 13]. The methodology is to use a fast simulation to position a workload at an interesting point (after booting, loading target software, etc.) and taking a checkpoint (they call it a "snapshot") of the state at that point (or multiple points) of interest. The checkpoint is then used to start a number of differently configured detailed simulations, allowing efficient parallel exploration of the architecture space. Compared to switching a model between abstraction levels during a simulation run, this offers a simpler, more robust and more efficient mechanism.

1.1. Checkpointing Implementation Issues

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

* Also PhD student at the Dpt. Microelectrònica i Sistemes Electrònics Universitat Autònoma de Barcelona, Barcelona. Spain.

1. 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.
2. Saving and restoring the simulation kernel state, such as event queues and the current simulation time.
3. Saving and restoring the simulation configuration in terms of which simulation models form part of a virtual platform, and how they are connected.

We have addressed these problems for hardware device models written in SystemC™, using the OSCI reference SystemC simulator version 2.2.0. To create a complete system involving processors, memories, and software, we used Virtutech Simics [4] to provide us with the rest of the platform, as well as a proven checkpointing infrastructure. That let us focus on the core technology issues rather than checkpoint file structure and disk I/O.

2. Simics Checkpointing Basics

To understand our implementation, some background on Simics is needed. Virtutech Simics [4] has implemented checkpointing for about ten years, based on a simple but powerful mechanism called attributes [2].

Each simulation model in Simics defines its own set of attributes, which are expected to define the entire model state needed to continue the simulation from a particular point.

The attributes are registered with the Simics kernel by the model source code at simulation startup, and have a name and a type. The attributes are set and retrieved from the simulation kernel using an interface distinct from the transaction-level interfaces used for memory accesses and device-device communication. Figure 1 shows a simplified view of device models and connections between them in Simics.

Attributes do not have a one-to-one correspondence with the implementation of the state of the model. Internal caches and data structures are often used for efficient simulation, but they are normally represented as simple values in a checkpoint. An important point to note is that since this makes the model state indirect with respect to the implementation, different implementations of a model can load the same set of attribute values. This makes checkpoints independent of model implementations, and it is quite common for model implementations to change while attribute sets and checkpoints remain constant.

Simics also represents the configuration of a system by means of attributes, converting pointers between simulation models into names, and storing names and types of model instances in checkpoints.

By using the Simics platform, we have been able to focus on the particulars of how to save and restore the state of SystemC device models, and not deal with how to put information in a file or the complexities of checkpointing the state of a processor model (as we use Simics existing processors and system models as the framework to run target software interacting with our SystemC models).

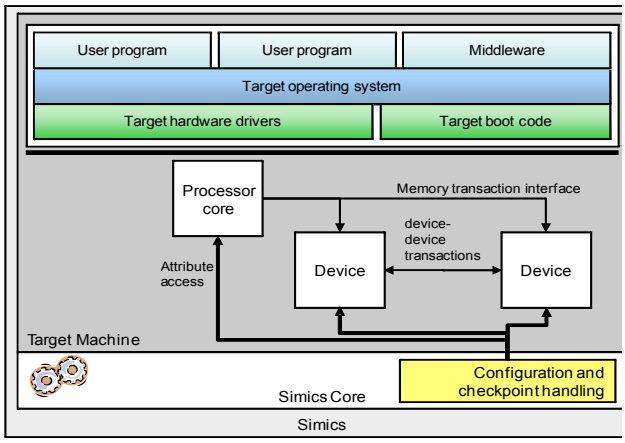

Figure 1: Simics back-door attribute system

When an attribute is accessed in Simics, the simulation module gets a call to a special attribute getter and setter functions. We used this mechanism to export the state of our SystemC models to Simics, and later get the saved values back. Thus, all our SystemC models need to do is provide a means to expose the state of models and the SystemC kernel to Simics.

Initially, we have considered the entire SystemC simulation setup as a single Simics "model", with a fixed internal configuration. Thus, we do not attempt to convey the SystemC simulation configuration in checkpoints, but rather consider that a fixed property of our simulation implementation.

3. SystemC in Simics

To run SystemC models in Simics, we made the SystemC kernel a slave system to the main Simics kernel, as shown in Figure 2. This system is called the *Simics-SystemC Bridge*.

The key problem in integrating two simulators is how to synchronize the time and execution of code in the two simulation worlds. In order to maintain simulation speed, we cannot synchronize SystemC with Simics on every target system clock cycle. Instead, we have designed a lazy scheme that only synchronizes when necessary.

When a transaction is sent into the SystemC part of the combined system (it could be a memory operation, reset, input, etc.), we call SystemC and lets the SystemC kernel and models process the transaction to completion. If SystemC has then progressed ahead of Simics time, we stall the Simics processor initiating the transaction to account for the time taken to complete the request.

After each transaction has completed, we check when the next event in the SystemC kernel is scheduled. If there is such an event, we post an event in the Simics event queue at a corresponding point in time. When the Simics event is triggered, the SystemC kernel is invoked so that time can catch up with Simics time and the event be processed. Thus, SystemC models can perform actions asynchronous to the Simics world, like serial console output or sending completion interrupts.

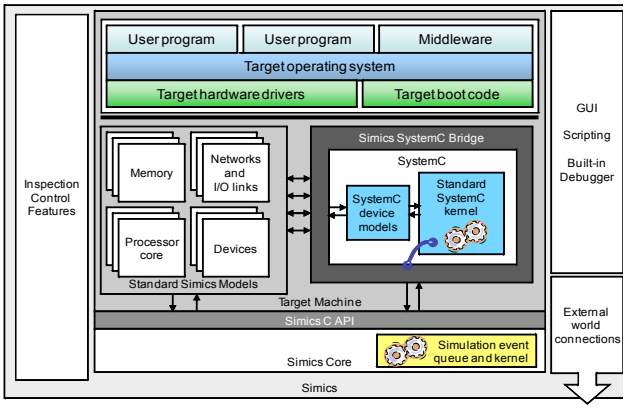

Figure 2: SystemC Encapsulated within Simics

The Simics-SystemC bridge also has to convert transactions between their Simics and SystemC types. This is a reusable feature for simple standard interfaces such as TLM-2.0 [6] or GreenBus [7], but for other interfaces you need a case-by-case conversion.

The performance of the integration is very good, as SystemC does not impose any slowdown on Simics when SystemC device models are not being activated. Also, the overhead of translating transactions is minor compared to the cost of actually computing the result of a transaction in a device model.

To handle checkpointing, the Simics-SystemC Bridge is extended to provide transactions corresponding to checkpoint save and restore. It also needs to convert from the state representation in SystemC to Simics attributes, and register attributes with Simics.

4. Checkpointing SystemC Model State

4.1. Model Requirements

Checkpointing requires a model to explicitly define the state that will be saved and restored. In our implementation, we have used the GreenControl [8] parameter mechanism to simplify the implementation in SystemC and make the changes to the SystemC code as small as possible.

GreenControl lets us mark variables in a SystemC module as managed parameters using the `gs_param<>` template. From 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, and even when the simulation is not running. Figure 3 shows a short example of the use of parameters. Essentially, this provides the back-door access we need to get and set the simulation state.

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

4.2. SystemC Parameters to Simics Attributes

There are two ways to connect the parameter handling in GreenConfig to the Simics attribute system, and we tried both.

Variant one is to create a Simics attribute for each parameter in the SystemC device. This makes the checkpoint very readable, and has the side benefit of exposing the state of the SystemC devices for easy modification and inspection from the Simics user interface. It suffers from the need to add code to define one attribute per parameter, however.

Variant two is to use the GreenControl feature of getting all parameters in a simulation as a list, and then convert that to a list attribute in Simics. This solution makes it somewhat more cumbersome to access the state from the Simics user interface as you have to parse a list of key-value pairs, but it also means that a single attribute is all that needs to be defined, reducing the amount of code in the integration.

Both solutions work equally well for checkpointing. The user-perceived difference is shown in Figure 4. We have used the list variant, as it is more flexible.

4.3. Limitations

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

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

However, note that this approach also means that the SystemC model implementation is separated out, and that

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

Figure 3: Example use of gs_param<>

```
simics> ckpt0->gs_all_param_value
"systemc_greencheckpoint_test.otherparm=42
;systemc_greencheckpoint_test.scparam=66;s
ystemc_greencheckpoint_test.scparam_two=47
11;"
simics> ckpt0->scparam
66
simics> ckpt0->scparam_two
4711
```

Figure 4: Simics CLI session with the two styles

different models can thus implement the same set of parameters. This supports such use-cases as changing the level of abstraction in a simulation and updating a model while reusing the same checkpoint.

5. Checkpointing SystemC Kernel State

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

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

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

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

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

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

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 reposted to the event queues. Thus, the SystemC kernel will run the simulation as if it were just continuing the checkpointed simulation without any interruption.

We access the event queue through the OSCI SystemC kernel class sc_simcontext. This class encapsulates SystemC kernel and it is in charge of managing the simulation. The list of events is communicated to Simics as a Simics attribute of the SystemC bridge module (Simics supports arbitrarily nested lists as an attribute type, which makes it easy to represent the event queue as a variable-length list). In order to access the list of active events and methods, we had to add some non-destructive inspection code to the OSCI SystemC 2.2.0 kernel.

When a checkpoint is taken in Simics, the event queue attribute is read and stored using the Simics checkpointing mechanism. When resuming, the Simics attribute is written with the value (old event queue state) stored in the Simics checkpoint file, and our SystemC bridge extensions then repost the events into the new SystemC simulation, recreating the state at the point when the checkpoint was taken.

6. Related Work and Alternatives

SimOS and IBM Mambo full-system simulator [3, 13] uses a checkpointing mechanism very similar to what we propose, with explicit model state separate from the implementation state. Typical uses for checkpoints in Mambo is to save complex system setups and to switch between different implementations and different abstraction levels.

The Boost C++ serialization library can save and restore the state of C++ objects in a program [9]. It provides portability and upgradability to new versions of the code. However, using this library with SystemC would have required a rewrite of the SystemC kernel to use serialized objects to store all state, and it is not clear how this would work with the cooperative multitasking nature of the SystemC kernel. The Boost library takes the same view as our solution on the state: only data stored in C++ object members are serialized, not temporary values on the stack, nor the state of threads in the system. Also, using Boost would require larger modifications to existing device models than our solution.

Another alternative solution that has been proposed is to save the entire contents of the memory of a running simulation process. At least in theory, this should work for any code, without modification, and including thread state. This is what it appears that CoWare and Cadence are doing in their recently announced of checkpoint support for SystemC, even if public details are quite scarce [10, 11]. The memory-dump solution has several limitations compared to our approach. As a checkpoint contains the state of the stack and heap, it is tied to the particular data layout and stack-frame layout of the code the simulation started with. Thus, it cannot be restored on a different machine (even a minor change such as a Linux kernel version or different set of system libraries can break it), nor can it be used with an updated or different ,model code.

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

7. Experiments

In our experiments with checkpointing, we have used a simple PowerPC-based virtual board containing a PPC603e processor, serial port, RAM, and a memory-mapped SystemC device. This system is sufficient to run code cross-compiled for a bare-metal target, and provide a focused platform for detailed experiments. A primary benefit of this simple system is that the target software can address the SystemC device directly as it has direct access to the entire memory of the target machine, which is much simpler than writing device drivers for a target operating system.

7.1. Simics-SystemC Bridge Performance

We tested the performance of the basic bridge itself using some by integrating an NS16550 serial port modeled in SystemC in a virtual development board based around a PowerPC 440GP SoC (an AMCC "ebony" board). The ebony board runs a full Linux kernel and U-Boot, and 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 microbenchmark program that pushed characters onto the serial port as quickly as possible, the performance was reduced by about five percent. Thus, we conclude that the SystemC bridge does not present a significant performance problem.

7.2. SystemC Checkpoint Support Overhead

In our SystemC implementation of checkpointing, there is a potential additional performance impact from the use of `gs_param<>`, since using the GreenControl mechanisms.

Preliminary tests on the performance of gs_param show results with about a 2% penalty against the same model without using internal variables modeled with gs_param.

7.3. 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 which is the contents of the RAM of the simulated machine containing code.

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

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

7.4. Validating Checkpointing

To validate that we can indeed save and restore a Simics simulation including a SystemC subsystem we used a fairly simple example device as shown in Figure 5. This device exhibits all essential problems for checkpointing, namely state in memory-mapped registers and an SC_METHOD sensitive to an event that is posted at some point in the future using sc_notify.

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

Our test consists of start the simulation and do a write to the register to start the periodic event. Then, do a checkpoint and quit simics. Then, we start simics again and we resume the simulation from the checkpoint. We could observe that the periodic event is triggering again as expected. With this test we validated that the SystemC kernel event list for SC_METHODs is properly saved and restored, and the simulation can be restored using our strategy.

7.5. Validating Model Updates

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

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

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

Figure 5: Code example

both registers from the start). Thus, we show that we can update models and use old checkpoints.

8. Discussion

This work has proved that it is possible to do checkpointing in SystemC, with a moderate effort and building upon existing frameworks. It has also exposed some issues in the SystemC design. The key problem is the provision of a Unix-style threaded execution model in SystemC, rather than an event-driven run-to-completion model. The threaded style encourages storing essential simulation state on the stack and as the location in the code, which is very hard to explicitly save and restore in a portable manner. Thank to the existence of `SC_METHOD`, a sound style can be implemented in the current SystemC framework.

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

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

9. Conclusions

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

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

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

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

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

In the future we will be investigating other approaches to checkpointing, and dealing with the case of SystemC threads. For example, it could be possible to converting threads to methods using automatic tools [12].

10. References

[1] M. Bergqvist, J. Engblom, M. Patel, and L. Lundegård, “Some Experience from the Development of a Simulator for a Telecom Cluster (CPPemu)”, *Proc. 10th IASTED* November, 2006

[2] Jakob Engblom: *Simics System Modeling*. Virtutech Whitepaper, May 2008.

[3] J.L. Peterson et al: " Application of full-system simulation in exploratory system design and development", IBM Journal of Research and Development, Vol 50, no 2/3, March/May 2006.

[4] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, B. Werner. “Simics: A Full System Simulation Platform”, *IEEE Computer,* Feb 2002.

[5] Jakob Engblom, “Modeling Language Produces TLM for Virtual Platforms”, *SCDSource.com*, Apr 9, 2008.

[6] *OSCI TLM-2.0 User Manual*, June 2008.

[7] Wolfgang Klingauf: *Systematic Transaction Level Communication Modeling with SystemC*, PhD Thesis, TU Braunschweig, Dept. of IC Design, 2008.

[8] Christian Schroeder, Wolfgang Klingauf, GreenControl. http://www.greensocs.com/en/projects/GreenControl

[9] R. Ramey: *Boost Serialization v.1.36.* http://www.boost.org.

[10] *CoWare Introduces First Ever Checkpoint/Restart Capability for Native SystemC Virtual Platforms*, April 14, 2008.

[11] Cadence. *SystemC Save and Restore Part 2 - Advanced Usage.* http://www.cadence.com/Community/blogs/sd/

[12] Robert Gunzel SCThreadConverter, http://www.greensocs.com/en/projects/SCThreadConverter

[13] M. Rosenblum and M. Varadarajan, SimOS: A Fast Operating System Simulation Environment, Stanford University technical report CSL-TR-94-631, July 1994.