

SIMICS
MODEL BUILDER

CHECKPOINTING

JULY 2009

JAKOB ENGBLOM

WWW.VIRTUTECH.COM

INTRODUCTION

Virtualized Systems Development[™] (VSD) is a development methodology where the actual hardware of a system is replaced with a virtual platform running on a workstation or PC. The virtual platform can run the same binary software as the physical hardware and is fast enough to be used as an alternative to physical hardware for software development.

The virtual platform provides additional benefits to the user compared to physical hardware. For example, the virtual platform offers superior convenience and stability, full insight into the system execution, and better debugging facilities. It provides access to the target system long before prototype hardware is available. This enables software development to start early. The virtual platform supports fault injection and testing with multiple configurations. It also provides checkpoint and restart facilities, which is the focus of this white paper.

Checkpointing is one of the key features of a VSD-enabled virtual platform. It allows work-flow optimizations and use cases that are impossible to achieve on physical hardware (or on simulators without checkpointing).

CHECKPOINTING

Checkpointing is the ability of a virtual platform to save the complete state of an executing simulation to disk (or host memory or some other suitable storage) and later bring the saved state back and continue the simulation from that point as if nothing had happened.

In general, checkpointing covers the following operations on the simulation:

- Saving the simulation state.
- Restoring the simulation state on the same host machine into the exact same compiled version of the simulation code.
- Restoring on a different host machine, possibly belonging to another user or organization (where different can mean a machine with a different word-length, endianness, operating system, and installed software base).
- Restoring into a bug-fixed version of the same simulation model.
- Restoring into a completely different simulation model that happens to have the same architected state.

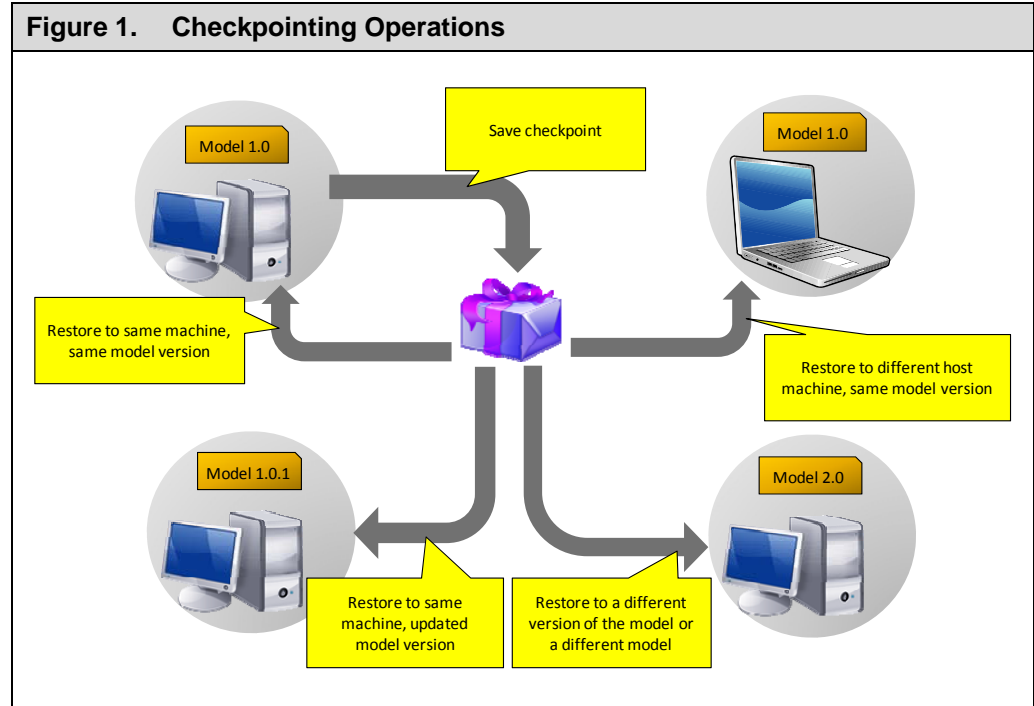


Figure 1 shows an overview of the four operations. In the remainder of this white paper, we will discuss when and why each type of operation applies, and how they are implemented in Simics.

A checkpoint includes the state of the simulated hardware. This covers processor registers, device registers, the current simulation time, the contents of memories and disks, and any network packets or other I/O in flight. The state of the software is implicit in the hardware state, as the software is really just code and data in processor registers and memory.

CHECKPOINTING USE CASES

Short-Term Saving of Work

The most obvious use of checkpointing is for a programmer to save the current state of a target system and bring it back up later as a way to save the work or enable undo of a risky operation. This would normally only require the ability to restore the same simulation on the same machine.

Avoid Repeating Boot and Target Configuration

With checkpointing, a repetitive target setup procedure can be done once and for all and saved. Subsequently, the fully configured system can be immediately resumed from that state, without any need to repeat the setup procedure. This gets more and more important as simulated systems increase in complexity and workload size. The most common case is booting an operating system, but in general, the system setup will also include loading target applications, starting servers, configuring networking software, and the setting of hardware configuration parameters by the simulation user and the target software. System boots can easily take hours of real-world time, as it involves the simulation of many billions of instructions across hundreds of processors.

Using checkpointing thus reduces the amount of time spent on routine work and increases the time spent doing real value-added work in an organization. Another benefit is that the state is consistent across multiple developers: since they all use the same checkpoint, there is less risk of a manual mistake in the setup process that would affect test results and cause spurious bug reports and wasted work.

This use of checkpointing requires checkpoints to be portable across hosts, since different developers can be expected to use different machines with different operating systems and processors. It is also very beneficial if checkpoints can be loaded into a different version of the simulation model, since this allows the booted system state to be reused across minor bug fixes to the hardware model.

Communicating System State

Checkpointing is also a powerful communications tool: it makes it possible for any user of a virtual platform to precisely communicate the system state and configuration to any other user. Instead of passing along instructions and snippets of code and a description of the initial target setup, a checkpoint can be sent that exactly and precisely describes the state of the system.

For example, a bug report from a test lab to a development department can be framed as “open this checkpoint, run for 10 ms, and watch the web server crash”. This assumes that the simulation is perfectly repeatable and that each time you open a checkpoint, the simulation will execute in exactly the same way (provided asynchronous input is controlled in some manner).

As is the case with reusing the booted state of a system, checkpoints need to be portable across hosts and at least minor model updates.

Communicating Model Bugs

Checkpoints can be used to communicate bugs found during the (iterative) development of a virtual platform. When users of a virtual platform find bugs in the platform hardware model, they can package up the simulation state (hardware and software) and pass it to the team developing the platform. The simulation team can then easily reproduce the perceived model bug, and validate if it is a bug in the hardware model or the software. If it is a hardware model bug, they can test any fixes they implement by updating the model, and opening the same checkpoint on the updated model. The checkpoint thus contains the test data necessary to validate the fix before sending an updated model to the software team.

A key enabler for this to work is the ability to open a checkpoint using an updated or completely different version of a model. The modeling team needs to be able to change the models, recompile them, and load the checkpoint taken on the old model into the updated model to verify that a fix works.

Parallelizing Testing

Virtual platforms offer a great way to parallelize the execution of system tests. Using clusters of server machines, many different test scripts for a system can be run in parallel, saving time compared to running them serially on physical hardware.

Checkpoints make this process easy, since the initial state for a test can be prepared on a developer's workstation and then transferred to the testing cluster. Copying the same checkpoint to multiple machines makes it possible to run many parallel simulations from the exact same initial state, trying different variations of tests.

Supporting Target System Training

When using a virtual platform to train operators and administrators for a system, checkpointing makes it very easy to recover a target system to a known good state. For example, if students using a Linux target system manage to destroy all user accounts or erase `/lib`, going back to a checkpoint taken before the actions makes undo trivial.

Checkpoints can also be used to save the state of a machine requiring user intervention to recover functionality. For example, a system where the target software stack is in an inconsistent state (such as a broken passwords file) or some alarms have triggered. By using checkpoints, instructors can reliably reproduce and repeat exercises with any student at any time, without needing to prepare a physical machine each time.

Using checkpoints in training requires portable checkpoints, as you do not want training setups to be bound to a particular machine or simulator version.

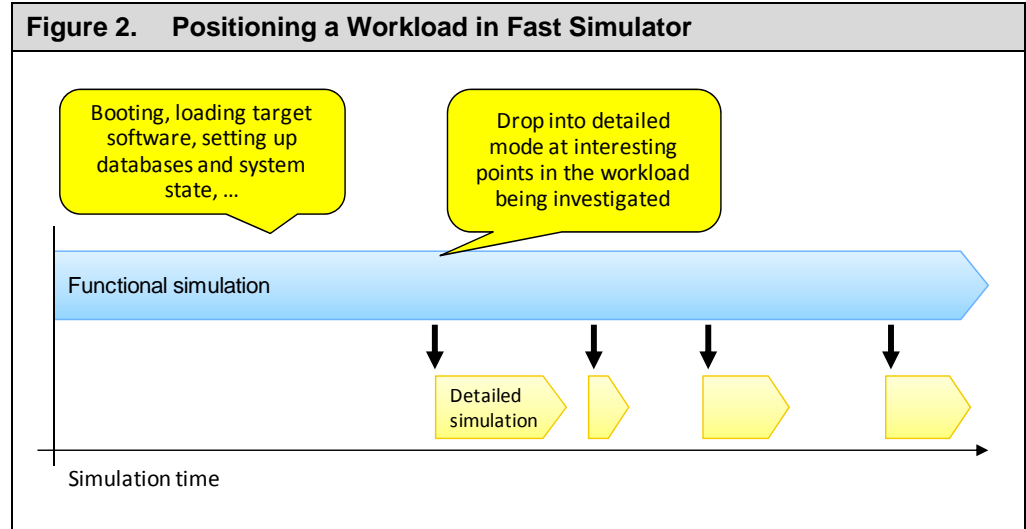
Changing Level of Abstraction

Checkpointing as we know it today first appeared in virtual platforms and full-system simulators in the mid-1990s¹. The primary use case at that time was to make it possible to change the level of abstraction, from a fast functional model to a slow detailed model, for the case of doing research in processor microarchitecture. This methodology has been successfully employed by all major processor design houses and computer architecture research groups for the past decade.

As illustrated in Figure 2, the idea is to run a system in a fast mode to boot it, load software, and position the software at interesting points in its execution. Then, take a checkpoint, and bring the checkpoint up into a completely different system model with much more timing and architectural details. This process can be automated and sampling theory applied to determine when to go into detailed mode, yielding what is known as sampled simulation.

A key requirement on the checkpoint system is that checkpoints should only capture the architectural state of a system, and not the implementation state of the simulation model. Thus, any model implementing the same architecture as seen from the software (for example, the same processor instruction set or device programming interface) can be used with the checkpoint. Checkpoints thus become portable across completely different implementations of the same system.

¹ An early mention of checkpointing used for this purpose is in the paper “SimOS: A Fast Operating System Simulation Environment”, by Mendel Rosenblum and Mani Varadarajan, Stanford University technical report CSL-TR-94-631, July 1994.



Note that in practice, the use of checkpointing to change the level of abstraction is a one-way process. Moving from a functional model to a detailed model is easy to support, as all that is communicated is the architectural state of the machine. The architectural state is a subset of the information in a detailed model, and thus it is easy to initialize a detailed model with the state from a fast model. Such an initialization obviously does not cover the state of caches or pipelines, and the most common solution to this problem is to run the detailed simulation for some time to warm up the microarchitectural state before starting to perform real measurements.

Checkpointing of detailed models is much more difficult to achieve. The state of a detailed model tends to be very large and convoluted due to the high complexity of the implementation. It is also hard to derive a precise architectural state from a detailed model, since the state in the detailed model will include queues, incoherent caches, and instructions and operations in flight that have not yet committed to an architectural state. Thus, at any instance in time, it is hard to define what the architectural state of the system actually is. It is sometimes possible to drain all queues and pipelines and move the system to a quiescent state, but that does not really achieve the desired instant checkpoint and restore of the detailed model. We know of no successful solution to checkpointing a detailed simulator of any non-trivial processor or system.

Archiving Target System Setups

In some cases, there is significant value in the work invested in just completing a target system setup. For example, a good configuration of

benchmark software setups can be very valuable for researchers and processor developers, and represent a significant investment in software compilation, configuration, and target system configuration. They represent a convenient packaging of target setup that can be reused any number of times from a known state, and can remain in use for several years as the starting point for benchmark runs and regression tests.

To be useful for such long-term archiving, checkpoints have to be portable across hosts, model versions, and model rewrites. Quite often, the checkpoint is used with a detailed model of a processor or hardware accelerator. The same experiment can thus be run with many different variants over time, without any difference in the target system setup affecting the results. For example, in the Simics academic computer architecture community and in some Virtutech Simics demo setups, checkpoints have seen lifetimes of five years or more, across several major versions of Simics.

IMPLEMENTATION ALTERNATIVES

Checkpointing can be implemented in a few different ways. In Simics, the solution is to have models explicitly export and import their state to and from the checkpoint. The model internal representation of its state can be arbitrary. The external state used in a checkpoint is a projection of the internal state, but has no necessary relationship to the implementation and how it stores the state of the simulation model.

This leaves models free to change the implementation and state representation between model versions. It also makes it possible for a model to have the same external checkpointable state as another model, supporting the change of abstraction levels, and the complete reimplementing of a model in a different way.

One alternative implementation is to store the state of the *simulation process* in its entirety, and revert to it later. This automatically stores all local variables, variables on the stack, and similar implementation state. This makes it virtually impossible to change the implementation of the model and still use the same checkpoint, or to move the checkpoint to a different machine (since that will in all likelihood change the memory layout even if it supposedly has the same OS and libraries available). The resulting checkpoint file is also very large, essentially the same size as the simulating process, rather than the essential state exported by a Simics checkpoint. Thus, this solution is considered too limited to be of practical use for full-system simulation.

An even more heavy-weight implementation is to run the simulation inside a *virtual machine* that supports snapshots, and use the VM snapshot to capture the state of the simulation as well as its operating environment. Such snapshots are very large, as they include the entire host memory as well as a fairly large disk image. They obviously do not support changing the models in any way, as the simulation program is kept running over the checkpoint process.

If Simics checkpoints tend to be on the order of megabytes, the process images tend to run to the hundreds of megabytes, and VM snapshots into multiple gigabytes. Overall, the Simics checkpointing system has proven to be the most flexible and powerful solution, enabling a very wide range of checkpoint uses.

Figure 3. Device Checkpoint Data Example

```
OBJECT argo0.soc.uart[0] TYPE NS16550 {
  queue: argo0.soc.cpu[0]
  component: argo0.soc
  component_slot: "uart[0]"
  object_id: "obj_0000008c9b51e4d4"
  build_id: 0xbc2
  irq_high: TRUE
  receive_ready_waiting: FALSE
  RXRDYn_high: FALSE
  TXRDYn: NIL
  interrupt_mask_out2: 0
  recv_fifo: []
  xmit_fifo: []
  receive_throttled: FALSE
  RXRDYn: NIL
  console: argo0.console0.con
  TXRDYn_high: FALSE
  in_fifo_mode: TRUE
  multiple_transfer: FALSE
  irq_dev: (argo0.soc.pic, "internal_interrupts")
  regs_ii: 1
  regs_scr: 0
  regs_msr: 176
  regs_ier: 15
  regs_iir: 2
  regs_mcr: 11
  regs_drm: 0
  regs_dr1: 161
  regs_lcr: 19
  link: NIL
  irq_level: 26
  trigger_lvl: 8
  xmit_time: 0
}
```

IMPLEMENTATION IN SIMICS

As stated above, the Simics implementation of checkpointing is based on the explicit import and export of model state from and to an external representation of the state.

In most cases, what needs to be exported is the *architectural* state of a device. This represents what is seen by the target software (such as programming registers), any defined operating modes and other device state, and how the model is connected to other models in the system. It is the responsibility of a model writer to define the necessary model state to export, but this is usually quite obvious from the definition of the device's software interface.

Figure 3 shows an example of the checkpoint state of an NS16550 UART used in many Simics target machines. Note that the checkpoint data contains some Simics-specific structural information like object IDs and build IDs. It also contains information on the other devices the serial port is connected to, using names like "argo0.soc.cpu[0]" and "argo0.console0.con" rather than pointers. This

is crucial to make checkpoints portable across machines and implementations. Some internal state is stored such as “irq_high” and the current contents of fifos. The user-visible registers are listed with their current contents.

All values are stored using host-independent bit-precise formats, including floating-point values (see Figure 5 for an example). Floating-point numbers cannot be saved in a standard decimal-point format, since the C libraries on different computer systems have proven to generate different actual values from the same input string.

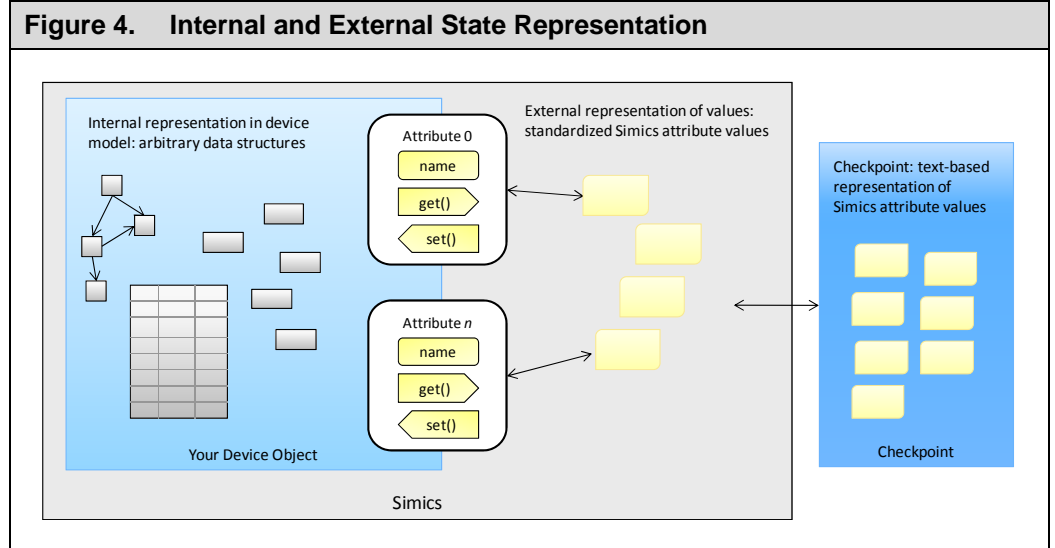
Memories and disks are quite special from a checkpointing perspective, as their state can be very large (memories of many gigabytes are common in virtual platforms, along with disks that span hundreds of gigabytes or even terabytes). To handle this, Simics uses a special “image” system that only saves the difference between one checkpoint and the next checkpoint (or between the initial state of a simulation and the first checkpoint), as files external to the main checkpoint file.

Thanks to differential saving, a checkpoint that boots a system from a multi-gigabyte disk image only needs to save the sectors of the disk actually changed during the boot. Multiple checkpoints can refer to the same fundamental disk image, as disk images are considered read-only by default. Checkpoints can also depend on other checkpoints, making it very efficient to save multiple snapshots from the same execution.

Checkpoints in Simics do not include the session state of the simulation, such as breakpoints, watchpoints, and active scripts. Recreating such session state is handled separately from checkpoints – the checkpoints focus on the state of the virtual platform. This separation of concerns has proven to be the most robust design over time.

Simics Attribute System

In Simics, the external state of a model is exported using *Simics attributes*. Each attribute has a name that identifies it and a type, and two functions: `get()` to read the value of an attribute and `set()` to change the value to something else. The type of an attribute is built from primitive types such as integers, boolean, floating-point, and strings, combined into lists or lists of lists. An attribute can accept several different input types, and lists can be of arbitrary length (and variable length).



As shown in Figure 4, the internal representation inside a device model has no necessary relationship to the external view. The `get()` and `set()` functions convert between standard Simics attribute value data structures, and the internal data structure of a device. Simics itself furthermore converts from the attribute values to the printed representation used in checkpoint files. Thus, devices never see strings they have to parse, only standard data structures exposed by Simics.

Having lists as a standard data type is crucial to the expressive power of Simics checkpoints. It makes it very easy to support changing and variable-size data structures, such as memory maps. Figure 5 shows an example memory map from an mpc8641 machine. Note how object names and mixed with integers and NIL values to mark “nothing” as the value for some parts of the map. Lists also make it possible to export the contents of C structs and C++ classes in an implementation-independent way, as the `set()` and `get()` functions convert from lists to structure members and vice versa.

Figure 5. Example of a List Attribute

```

OBJECT argox2_0.soc.ccsr_space TYPE memory-space {
  queue: argox2_0.soc.cpu[0]
  component: argox2_0.soc
  component_slot: "ccsr_space"
  object_id: "obj_000001b624ee96ba"
  build_id: 0xbc2
  map: ((0, argox2_0.soc.ccsr, 0, 0, 0x1000, NIL, 0, 8, 0),
        (0x1000, argox2_0.soc.mcm, 0, 0, 0x1000, NIL, 0, 8, 0),
        (0x2000, argox2_0.soc.mc[0], 0, 0, 0x1000, NIL, 0, 8, 0),
        (0x3000, argox2_0.soc.i2c[0], 0, 0, 24, NIL, 0, 8, 0),
        (0x3100, argox2_0.soc.i2c[1], 0, 0, 24, NIL, 0, 8, 0),
        (0x4500, argox2_0.soc.uart[0], 0, 0, 17, NIL, 0, 8, 0),
        (0x4600, argox2_0.soc.uart[1], 0, 0, 17, NIL, 0, 8, 0),
        (0x5000, argox2_0.soc.lbc, 0, 0, 0x1000, NIL, 0, 8, 0),
        (0x6000, argox2_0.soc.mc[1], 0, 0, 0x1000, NIL, 0, 8, 0),
        (0x8000, argox2_0.soc.pcie_bridge[0], 0, 0, 0x1000, NIL, 0, 8, 0),
        (0x9000, argox2_0.soc.pcie_bridge[1], 0, 0, 0x1000, NIL, 0, 8, 0),
        (0x21000, argox2_0.soc.dma, 0, 0, 0x1000, NIL, 0, 8, 0),
        (0x24000, argox2_0.soc.tsec[0], 0, 0, 0x1000, NIL, 0, 8, 0),
        (0x25000, argox2_0.soc.tsec[1], 0, 0, 0x1000, NIL, 0, 8, 0),
        (0x26000, argox2_0.soc.tsec[2], 0, 0, 0x1000, NIL, 0, 8, 0),
        (0x27000, argox2_0.soc.tsec[3], 0, 0, 0x1000, NIL, 0, 8, 0),
        (0x40000, argox2_0.soc.pic, 0, 0, 0x40000, NIL, 0, 8, 0),
        (0xc0000, argox2_0.soc.rapidio, 0, 0, 0x20000, NIL, 0, 8, 0),
        (0xe0000, argox2_0.soc.gu, 0, 0, 0x1000, NIL, 0, 8, 0),
        (0xe1000, argox2_0.soc.pmc, 0, 0, 0x1000, NIL, 0, 8, 0),
        (0xe2000, argox2_0.soc.debug, 0, 0, 0x1000, NIL, 0, 8, 0),
        (0xffff00, argox2_0.hfs, 0, 0, 16, NIL, 0, 8, 0))
  timing_model: NIL
  snoop_device: NIL
  default_target: NIL
}

```

Simics attributes are also used as a way to support inspection of a target machine at run-time. Given additional meta-data, they are used to provide device register lists with information about register names and documentation, to present the memory map of a setup in a graphical browser, and many other uses. It is a simple general mechanism that can be used in many different ways.

Checkpoint-Friendly Modeling

To support checkpoint and restore, simulation models need to follow a few simple rules (which apply to all simulation frameworks that support checkpointing to the same extent as Simics, and are not really Simics-specific).

Models have to define the state they expose in a checkpoint to allow a complete recreation of the simulation. Models should also allow the

simulation state to be set to anything at any point in time, and not expect that data stored in local variables survive between two transactions occurring.

Models should handle time by querying the simulation kernel for time anytime it is needed. Models should only depend on virtual time for their execution, as the host time can change arbitrarily (for example, a checkpoint being opened several years after it was saved).

Models should not refer directly to resources on the host machine, as that prevents portability and introduces non-deterministic behavior. For example, if a model wants to load test data from files, it should be stored inside the Simics simulation in images so that it can be checkpointed along with the rest of the simulation (a simpler approach is usually to let Simics handle the loading of test data into target memory, and read it from target memory).

To access host networks, user interface consoles, and similar items external to the simulation, models should use the facilities present in Simics that provide indirection between a model and the host machine and user. This puts the tricky code that handles checkpointing of user interfaces and network connections into a single place, and makes it unnecessary for models to take care of such issues themselves.

Models have to be written in an event-driven style, where all timed activation of a model is implemented using timed events posted to the simulator kernel. This provides the simulator with the information it needs to save and restore the time behavior of a model. Threads are

In practice, Simics models developed using the DML toolkit² automatically support checkpointing and reverse execution without any extra work for the device modeler. Attributes are automatically generated where appropriate to store model state.

Checkpointing non-Simics Models

Simics checkpointing can be used to wrap state export/restore functionality found in simulation models not developed for Simics checkpointing initially. The only requirement is that the simulation model provides some way of exporting and importing its internal state. The data from the non-Simics-

² See http://www.virtutech.com/whitepapers/virtutech_dml.html for more details on DML.

native model is usually represented as a single Simics attribute that takes whatever the model exports, and converts into a Simics list of values or a raw data attribute. In this way, Simics system models can leverage existing checkpointable models without requiring the models to be rewritten.

Checkpointing and Threading

The use of any kind of threading in a model makes checkpointing very difficult to achieve in practice. In Simics, threading is thus not supported in the checkpointing mechanism or in the basic Simics device modeling API. The problem with threads is fundamental, in that they are an *implementation* mechanism, not a part of the target system state.

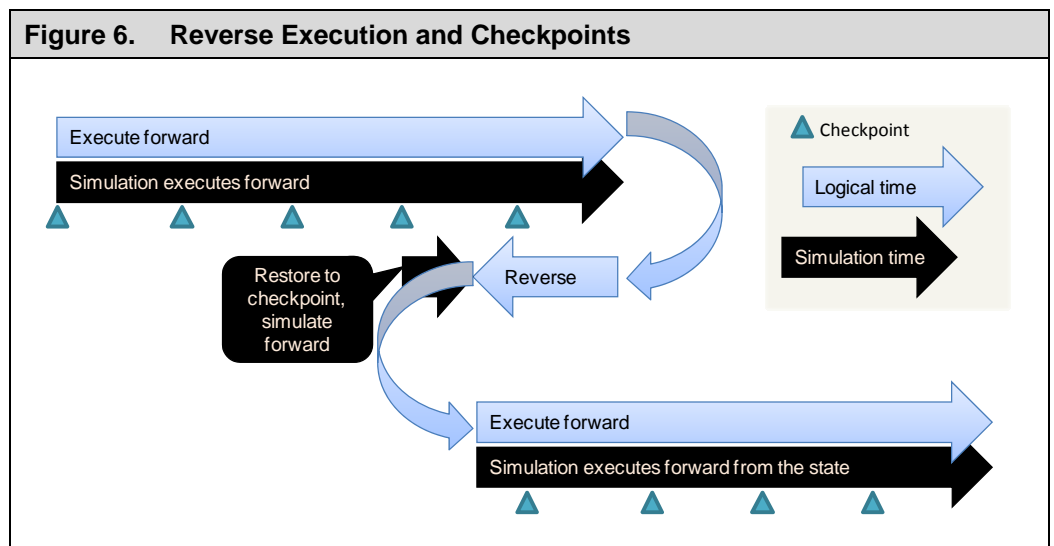
If you want to restore the state of a *model* to where it left of, and that model contains threads, you really expect to see the threads to the same state. This includes the state of their local call stack, registers with local variables, the program counter, and the stack pointer. Such information cannot be saved in a portable and reliable manner, as it is directly derived from a particular compilation of a particular version of the code. The smallest change to the code or the addition of a single variable will throw the mechanism off. Not to mention modifying a model to add threads, remove threads, or use threads in a different way.

A mechanism that converts the current thread location to an explicit state variable and then checkpoints that state is certainly possible to create. However, that is fairly complicated to implement, and an event-driven model will achieve the same with less complexity in implementation. Another advantage of a pure state variable and event-driven approach is that the state of the model can be investigated at any point in time, and potentially changed.

Note that some Simics modules do use host-operating-system threads for their implementation. Such modules are special cases and special care is taken when creating them to support checkpointing. Usually, they correspond to special functions that interact with the host or the user, such as network connections to the real network or graphical displays.

REVERSE EXECUTION

Checkpointing is also the basic mechanism behind reverse execution in Simics and other simulation tools and virtual machine tools³. In essence, the logical time of a simulation can be reversed and the simulation made to seem like it is running backwards in time by going back to an earlier checkpoint, and executing forward until just before the current time.



This obviously requires checkpoints to be very fast to create and restore. In Simics, models are not recreated every time simulation restarts, but rather the state of the existing set of models is updated.

This puts some additional requirements on simulation models. The models have to accept a new state at any point in time, not just once as for basic checkpoint and restore. The models need to handle time in such a way that they work even when the time in simulated time that a transaction occurs can be arbitrary and with no necessary relationship to the previous transaction seen. Any handling of time has to be done with the simulation kernel, so that it can handle and change the simulation time freely.

³ In particular, VMWare Workstation 6.0 and later supports snapshot, record/replay, and reverse debugging using this mechanism.

SUMMARY

Checkpointing is a very powerful feature of Simics virtual platforms. With checkpoints, users can save their work and share target system state and bugs between them. Implementing checkpoints requires some care when building models, but the care pays off as a model evolves and a virtual platform is used across development groups and system development phases. With checkpointing, many tasks can be automated or simplified, saving developer time and building better systems faster.

CONTACT INFORMATION

North and South America sales_americas@virtutech.com	Europe, Middle-East, Africa sales_emea@virtutech.com
Asia-Pacific sales_apac@virtutech.com	Japan sales_japan@virtutech.com
http://www.virtutech.com	

© Copyright 2009 Virtutech, Inc. All Rights Reserved.

TRADEMARKS. Virtutech, Simics, Hindsight, and the logos thereof, are trademarks or registered trademarks of Virtutech, Inc. and/or its subsidiaries, in the United States and/or other countries.

THIS PUBLICATION IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT.

THIS PUBLICATION COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION HEREIN; THESE CHANGES WILL BE INCORPORATED IN NEW EDITIONS OF THE PUBLICATION. VIRTUTECH MAY MAKE IMPROVEMENTS AND/OR CHANGES IN THE PRODUCT(S) AND/OR THE PROGRAM(S) DESCRIBED IN THIS PUBLICATION AT ANY TIME.

Virtutech, Inc., 2001 Gateway Place, Suite 201E, San Jose, CA 95110, USA