Checkpointing SystemC Models

Jakob Engblom, Virtutech

Màrius Montón and Mark Burton, GreenSocs

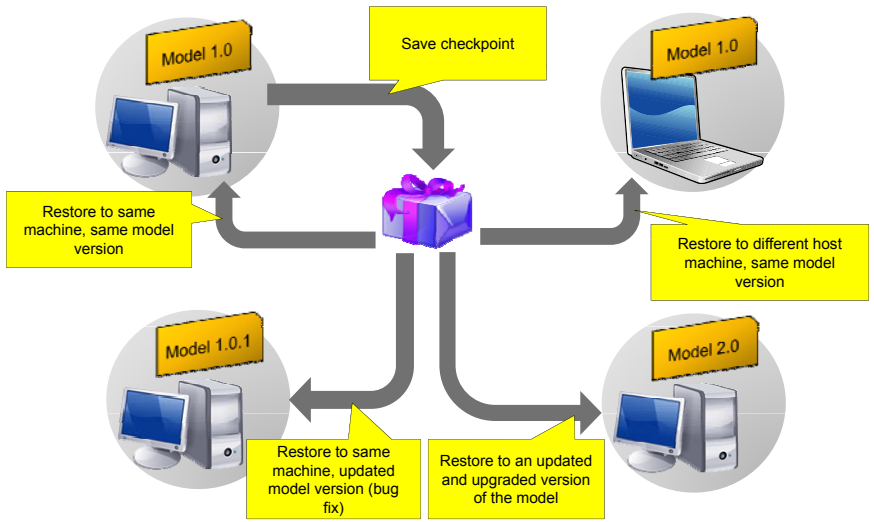
WHAT IS CHECKPOINTING?

Checkpointing

- **The ability to run a simulation to some point**
- **Save the simulation at that point**
- **Quit the simulation**
- **Start a new simulation, and pick up from the checkpoint**
  - Continuing to simulate at the exact point in simulation time where we left off
  - With the complete simulation state
- **Across model version**
- **Across hosts**
- **Across simulator versions**

Also see http://www.virtutech.com/whitepapers/simics_checkpointing.html

Checkpointing Operations



Implementing Checkpointing

- **Models explicitly expose their state**
  - Explicit operation to convert from internal state used during simulation to external state stored
  - Explicit operation to convert from stored state to internal state
- **The checkpointed state should basically be the architectural state of the hardware**
  - In principle, independent of the model implementation
  - Useful at any level of abstraction
  - Cannot avoid some simulation-specific artefacts, in practice

Example from a checkpoint of a serial port

```
OBJECT argo0.soc.uart[0] TYPE NS16550 {
                    queue: argo0_cpu0
                    build_id: 0x9cb
                    irq_dev: (argo0_pic, "internal_interrupts")
                    recorder: argo0_recorder0
                    link: NIL
                    console: argo0_con0
                    xmit_fifo: ()
                    rcvr_fifo: ()
                    rbr: 0
                    rbr_busy: 0
                    interrupt_requests: 0
                    interrupt_pin: 0
                    thr: 0
                    ier: 5
                    iir: 193
                    fcr: 129
                    lcr: 19
                    mcr: 11
                    lsr: 96
                    msr: 48
                    scr: 0
                    dl: 0x6b7
                    xmit_time: 0
                    waiting_for_tx: 0
                    waiting_for_tx_fifo: 1
                    overrun_debug_level: 2
                    target_pace_receive: 1
                    irq_level: 26
                    interrupt_mask_out2: 0
}
```

This is in Virtutech Simics, in a native Simics model, but it shows the principle. The implementation language does not matter.

Examples of Checkpointing Use Cases

- **Save your work**
  - Just like “save” in a word processor
- **Avoid repetitive simulations**
  - Immediately go to a booted OS, configured network fabric, etc.
- **Communicate system state**
  - Send checkpoint from test dept to software development dept
  - Includes hardware and software, trivial to reproduce bugs
- **Communicate model bugs**
  - Give model developers a test case, test that model works when updated
- **Parallelizing simulation work**
  - Start many simulation from the same setup state, vary local parameters
- **Change level of abstraction**
  - Use fast TLM models to setup system state, store to checkpoint
  - Open in a detailed simulator
  - Since we store the architectural state, this is fairly easy operation
- **Archive target system setups**
  - In particular, complex software setups

Checkpointing is totally addictive. Once you have seen it work, you never want to do without it again.

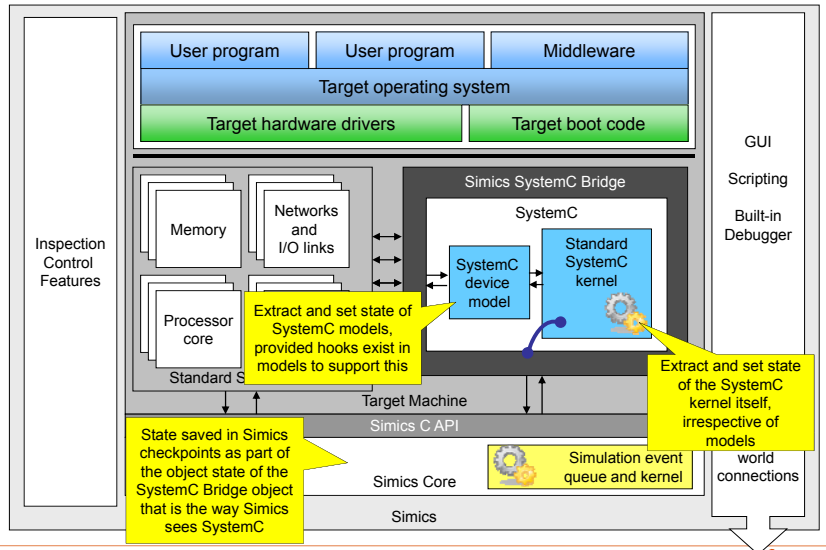
Theoretically Alternative Implementations

	Use standard VM snapshotting system	Dump simulation process to disk
What	Store a VMWare (etc.) snapshot of a complete workstation session	Store the contents of the Simics process’ memory to disk, bring it back up later
Advantages	Does save any and all state in the simulation. No need to change models to checkpoint.	Smaller than VM snapshot. Does save any model state with no need to change models.
Disadvantages	Very large (many GBs) Very slow to take a snapshot Does not support updating models and retrying from a checkpoint Not portable across hosts Not portable across model versions	Quite large (100s of MB) Does not support updating models and retrying from a checkpoint Not portable across hosts Not portable across model versions We tried this once in Simics, and it just fell apart in practice. See also: http://jakob.engbloms.se/archives/817

CHECKPOINTING SYSTEMC

Using Simics as the infrastructure

SystemC in Simics with Checkpointing



Saving Model State in SystemC Models

- **Requires model to explicitly define the state**
- **Requires models to accept an update to their state after they have initialized (i.e., post-elaboration)**
  - SystemC bridge creates the complete simulation model setup
  - SystemC bridge then updates the SystemC time and event lists from the save
  - In a separate step, goes through parameters and changes values in model
  - This state update happens post-elaboration in SystemC terms
    - Would be nicer to do pre-elaboration, but that requires redesigning SystemC
- **Requires model to adhere to coding guidelines**
  - More later on this

Marking State in SystemC Models

- **Using GreenSocs GreenConfig**
  - Configuration library for SystemC
  - Declare "parameters" in SystemC code
  - Parameters behave like regular variables in the code
  - Parameters have back-door access to retrieve and change their value
- **Entire SystemC model state exported to a single Simics attribute**
  - Using GreenConfig to/from string ability
- **Assumes SystemC model setup is constant from run to run, not saved in checkpoint**

Code Example

```
// From timer_greencheckpoint/timer.h
// state: programming registers
gs::gs_param<gs_uint32 > register_control; // bit 0 is IE
gs::gs_param<gs_uint32 > register_status;  // bit 0 is OC
gs::gs_param<gs_uint32 > register_bell;
gs::gs_param<gs_uint32 > register_countdown;

// state: interrupt high or low?
gs::gs_param<bool> interrupt_status;   // shadows OC,
mostly
```

Exported to Simics Attribute

```
simics> tgc0->gs_all_param_value
"timer_greencheckpoint.interrupt_status=0;timer_gree
ncheckpoint.register_bell=1;timer_greencheckpoint.re
gister_control=1;timer_greencheckpoint.register_coun
tdown=100000;timer_greencheckpoint.register_status=0
;"
```

http://www.greensocs.com/en/projects/GreenControl/GreenConfig

Demo Timer Code Snippets

Memory Operation Decode

```
int timer_greencheckpoint::IPmodel(accessHandle t)
{
  data.set(t->getMData());
  uint32_t op_addr = t->getMAddr();
  if (t->getMCmd()==Generic_MCMD_RD){
    // Read command incoming!
    switch(op_addr) {
      // Countdown  -- normal read semantics
      case countdown_offset:
      (*(gs_uint32*)data.getPointer()) =
        register_countdown;
      break;
  …
 } else if (t->getMCmd()==Generic_MCMD_WR) {
    switch(op_addr) {
      case control_offset:
        if(1 ==(value & 0x01)) {
          // Interrupts enabled!
          register_control = 1;
        } else {
          register_control = 0;
          // lower any interrupt pending
          if(interrupt_status == true) {
            interrupt_status = false;
            intr = false;
  …
```

Setting up Timer Event

```
// Write to countdown register

  case countdown_offset:
    // 1. If any old timer was still pending, cancel it:
    if (register_countdown != 0) {
      timer_event.cancel();
    }

   // 2. Set new value
   register_countdown = value;

   // 3. Check if we are beginning a count-down:
  if(register_countdown != 0) {
    // Set register flags:
    register_status = 0;  // not complete yet

   // Post event for delayed work
   timer_event.notify(sc_time(register_countdown, SC_US));
  }
```

Limitations: SystemC Constructs and Checkpointing

- **Checkpointable**
  - `SC_METHOD`
  - `sc_event`
    - Automatically checkpointed
  - `sc_int, sc_uint`, etc.
    - Just data types
    - Templated using GreenConfig
  - Basically, properly written efficient TLM models can be checkpointed with ease
  - Checkpointing usually infeasible for cycle-detailed models: too much intricate state to untangle and set
- **Not checkpointable**
  - `SC_THREAD, SC_CTHREAD`
    - State on the stack and in program counter, cannot be retrieved and set
    - Ties checkpoint to implementation, which is a complete no-no
  - `wait()`
    - Only meaningful with threads
  - `sc_mutex, sc_semaphore, sc_buffer`
    - Only meaningful with threads
    - State cannot be accessed
  - `sc_signal, sc_fifo`
    - State hidden inside kernel, but that might be fixed
    - Not very TLM-friendly

Limitations: Model Coding Guidelines

- **Workaround for limitations on sc_signal and sc_fifo:**
  - Use a checkpointed variable to mirror signal state
  - Drive the signal value from the variable value
  - sc_signal just for value movement

Workaround example

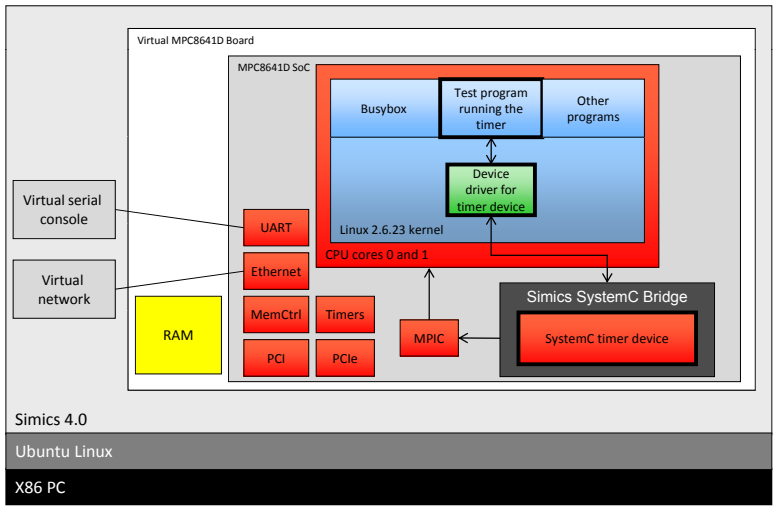
```
...
sc_signal intr;
gs_param<bool> interrupt_status;
...

// lower any interrupt pending
if(interrupt_status == true) {
  interrupt_status = false;
  intr = false;
  cout << "(SystemC) Also lowered interrupt line"
       << endl;
}
```

- **Time handling:**
  - Use timed events to drive simulation
  - Convert continuous wait()-driven code to event-driven code
- **Value changes**
  - Model has to accept value changes at any point in time, and still function
  - In general, requires callbacks on parameter changes

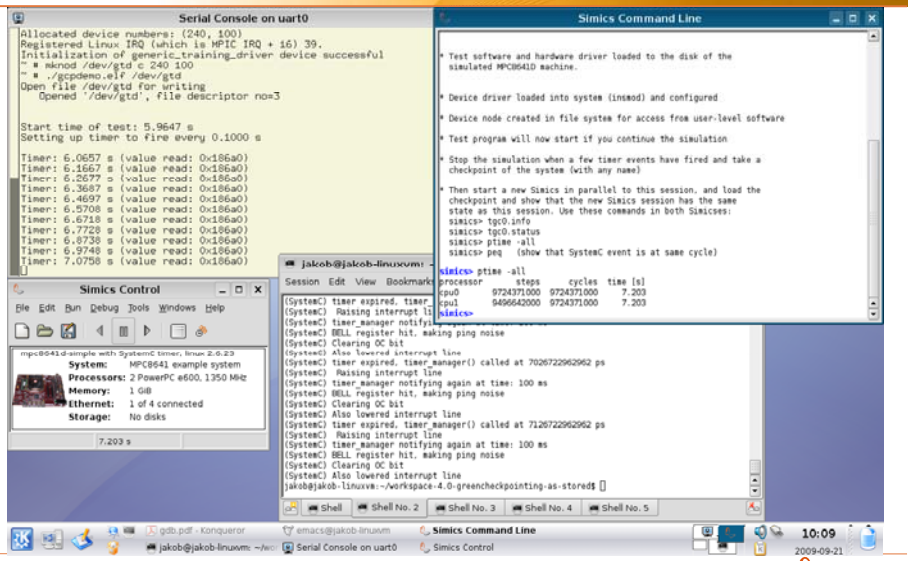
Getting to Kernel State

- **OSCI SystemC kernel not really friendly to inspection and checkpointing**
  - No way to non-intrusively extract and set the queue of events in the kernel
    - Note that checkpointing has to be non-destructive: the simulation should continue in the current simulation
  - No way to extract and set the state of signals, fifos, and other channels
- **Solution:**
  - Modify the kernel source code
  - Some ”friend” declarations to get to hidden state in the C++ type system
  - `sc_event.h`:
    - `sc_event, sc_event_timed` friend with our checkpointing handling class
  - `sc_pq.h`:
    - Added function to get the process queue without changing it

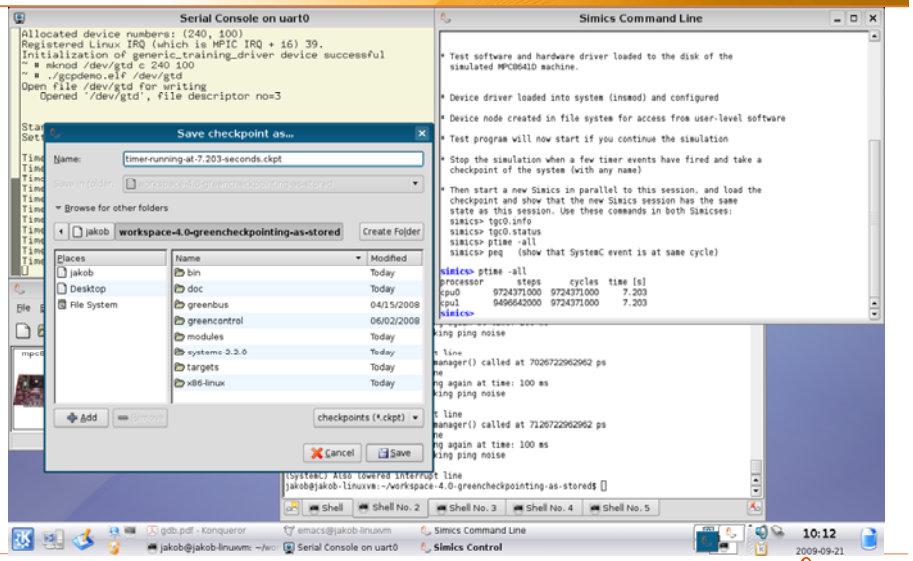
Test Setup



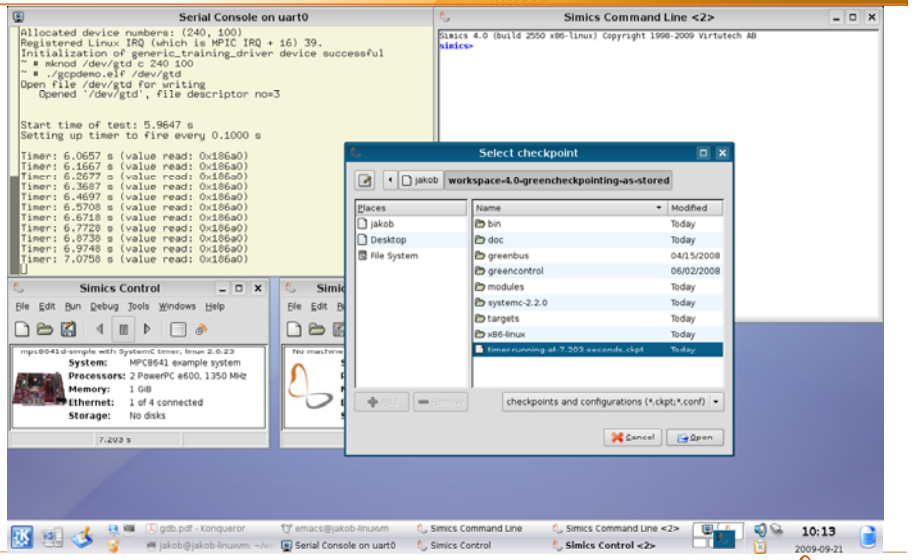
How it works



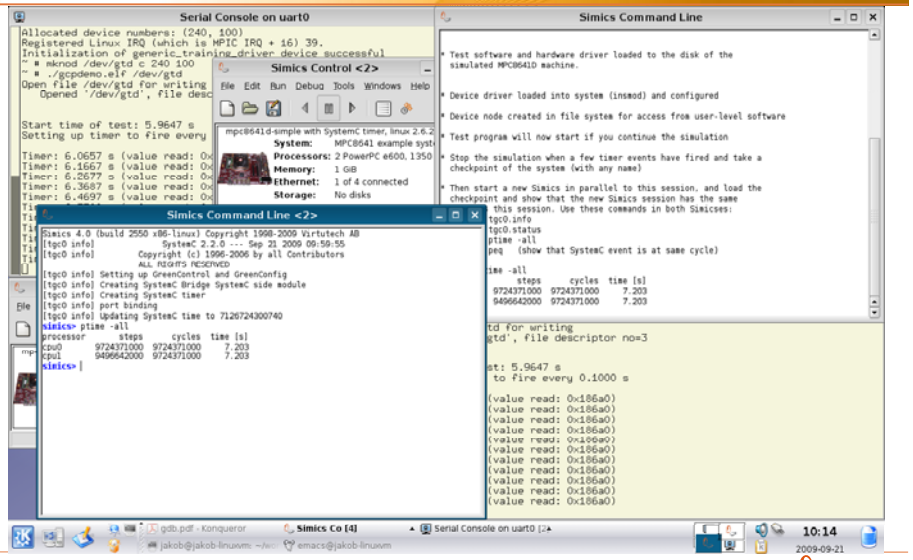
How it works



How it works



How it works



Future Work

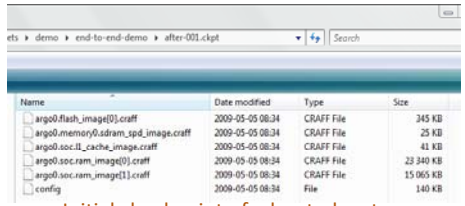
- **Extend checkpointing more parts of SystemC library**
- **Use forthcoming SystemC configuration libraries**
  - OSCI CCI WG is producing something quite useful
- **Lobby for SystemC improvements**
  - Concept of explicit device state as opposed to implementation state
  - Abolish unnecessary concept of *elaboration* and simulation *phases*
  - Outlaw threads from SystemC

THANK YOU!

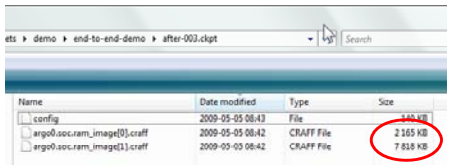
BACKUPS

Incremental Disk Images

- **Simics "image" class:**
  - Always 64-bit references
  - Use of memory far larger than host memory
  - Lazy allocation of host memory
  - Optimized swapping to disk
  - Backing store for RAM, FLASH, ROM, disks, any other form of bulk storage
- **Simics tracks changes to images**
  - Only changes and memory areas actually in use are stored in checkpoints
  - Changes since last checkpoint (or start of system)
- **CRAFF file format**
  - Compressed Random Access File Format, Virtutech-designed
  - Use "craff" utility in Simics to convert to other formats



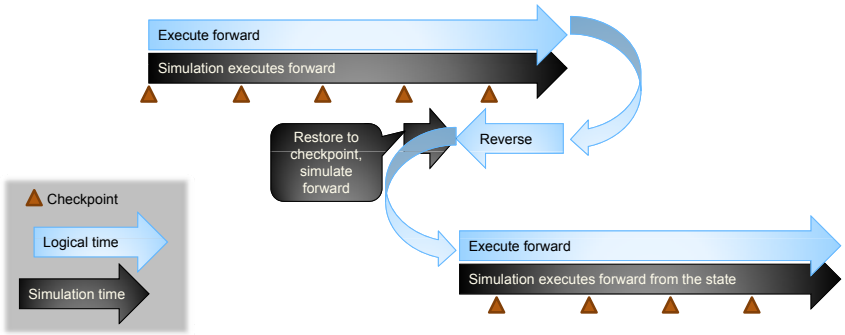
Initial checkpoint of a booted system



After loading some software on it

Reverse execution in Simics

- **Take periodic checkpoints of system state as we execute**
- **To go back to a point in time**
  - Go back to the closest checkpoint and execute forward



Why not Threads?

- **State in inaccessible places**
  - Program counter (where in the thread it is waiting for activity to happen)
  - Stack pointer register
  - Processor registers (local variables)
  - Program stack (stack-based variables)
  - Note that other object-oriented "serialization" solutions like boost::serialize and various Java libraries all just save object state and not thread state
- **Recreating such state is even harder than accessing it**
  - Need to setup a valid call-stack
  - Setup processor registers, program counter, stack pointer
- **Any change to program will change the layout**
  - Different program counter for the same statement
  - Different register allocation
  - Different stack layout for function
- **Even if no variables are "important"**
  - Still need to at minimum to force threads to the right wait() spots
  - Which is essentially as difficult
  - You cannot avoid this, as where threads wait() affect what is going to happen in the model