WIND RIVER

Wind River® Simics®

APPLICATION NOTE

Understanding Simics Timing

4.6

Revision 4081
Date 2012-11-16

Corporate Headquarters
Wind River
500 Wind River Way
Alameda, CA 94501-1153
U.S.A.

Toll free (U.S.A.): 800-545-WIND
Telephone: 510-748-4100
Facsimile: 510-749-2010

For additional contact information, see the Wind River Web site:
`www.windriver.com`
For information on how to contact Customer Support, see:
`www.windriver.com/support`

Chapter 1

Introduction

Running a program in Simics is not the same thing as running it on real hardware in the general case. Simics should be viewed as one possible implementation of the architecture, where physical hardware is another. They both comply with the hardware architecture, as defined by the architecture documentation, but the implementations will differ and one of the most important differences is timing. One could differentiate between the timing and the function of a system. The function of a system can be thought of as the state changes the system goes through, given the current state and inputs. These state changes are usually described in a programmer's reference manual or similar document, and they need to be modeled correctly by a simulator. The timing on the other hand is when these state changes occur. It is common that this is not defined by the architecture and thus, there is some headroom for the simulator to make its own choices.

The timing may differ between hardware and simulation model for a number of reasons:

- The timing of the system modeled may be undocumented.
- To increase simulator performance.
- To simplify the simulation timing model, so that a user fully understands it, should he want to alter it.

In Simics the most evident timing difference compared to real hardware is instruction timing. In real hardware, the instruction timing depends on memory access latencies, the complexity of the operation, execution resource contention, etc. These things are not simulated by Simics by default. Instead, one instruction is normally simulated each cycle. The simulation of the execution of one instruction is called a *step*. It is possible to configure a Simics processor to execute more or less than one step per cycle. It is also possible to have two or more processors in one simulation, where each processor has a unique clock frequency. In multiprocessor simulation time advances on one processor at a time, in a round robin fashion. Each processor advances a fixed amount of simulated time, called a *quantum*, before switching to the next processor. Even though each quantum has the same duration in simulated time, processors may simulate a different amount of steps, depending on their frequency.

Simics offers the possibility to stall memory transactions. This is a common way to affect instruction execution timing to make the simulation more realistic. Memory transaction are usually initiated by a processor and sent to a *memory space*. The memory space will in turn decide where the memory transaction should go based on its mappings. Each memory space provides the *memory hierarchy interface*. This interface provides a way to inspect and even alter a memory transaction passing through the memory space, both before and after the actual transaction has executed. In addition, the memory hierarchy interface lets a user stall the transaction. This is how caches are modeled in Simics. It is important to know that all memory transactions are not visible through the memory hierarchy interface by default. During normal operation, instruction fetches are not visible at all, and some data access are not sent to the memory space. Simics caches the result of a memory space look-up so that further access to the same area does not have to do the look-up again. This is done to increase performance. However, Simics can be configured so that all memory transactions are visible through the memory hierarchy interface.

Chapter 2

Events

Simics is an *event driven simulator* with a maximum time resolution of a *clock cycle*. In a single-processor system, the length (in seconds) of a clock cycle is easily defined as 1 divided by the processor frequency set by the user. As described later in this chapter, Simics can also handle multiprocessor systems with different clock frequencies, but we will focus on single-processor systems for the rest of this section.

As Simics makes the simulated time progress, cycle by cycle, events are triggered and executed. Events include device interrupts, internal state updates, as well as *step* executions. A *step* is the unit in which Simics divides the execution of a flow of instructions going through a processor; it is defined as the execution of an instruction, an instruction resulting in an exception, or an interrupt. Steps are by far the most common events.

Steps and cycles are fundamental Simics concepts. Simics exposes two types of events to users: events linked to a specific step, and events linked to a specific cycle. Step events are useful for debugging (execute 1 step at a time, stop after executing 3 steps, etc.), whereas time events are rather independent from the flow of execution (sector read operation on the hard disk will finish in 1 ms).

For each executed cycle, events are triggered in the following order:

1. All events posted for this specific *cycle*, except step execution.
2. For each step scheduled for execution on this cycle:
   (a) All events scheduled for that specific *step*.
   (b) A step execution.

Events belonging to the same category are executed in FIFO order: posted first is executed first.

Chapter 3

Instruction Execution Timing

Simics in-order

In the default model, the execution of a step takes no time by itself, and steps are run in program order. This is called the Simics *in-order* model. It implements the basic instruction set abstraction that instructions execute discretely and sequentially. This minimalistic approach makes simulation fast but does not attempt to model execution timing in any way.

Normally one step is executed every cycle, so that the step and cycle counts are the same. See the section **Changing the Step Rate** for how to change this.

Stalling

The in-order model can be extended by adding *timing models* to control the timing of memory operations, typically using the memory hierarchy interface described in the *Extension Builder User's Guide*. When timing models are introduced, steps are no longer atomic operations taking no time. A step performing a memory operation (whether an instruction fetch or a data transaction) can stall for a number of cycles. Cycle events are performed during the stalling period as time goes forward. Step events are performed just before the step execution, as in the default model. Simics executes one step at a time, but with varying timing for each step, so the simulation is still performing an in-order execution of the instruction flow. The basic step rate can also be changed; see the section **Changing the Step Rate** below.

Choosing an Execution Mode

Choosing an execution mode is matter of trade-off between performance and accuracy. The stalling mode is notably slower than in-order, but allows for memory timing models to operate correctly and, as a consequence, also allows for inspection of all memory transactions. By using checkpoints, it is possible to switch between the two modes, since a checkpoint created in the in-order mode can be loaded in stall mode. The simple in-order mode can thus be used to reach interesting parts of the simulation quickly, and the stall mode is used to simulate those parts.

Changing the Step Rate

The *step rate* is the number of steps executed each cycle, disregarding any stalling. It is expressed as the quotient q/p. By default, $p = q = 1$; this schedules one step to run in each cycle. This can be changed by using the ⟨**processor**⟩**.set-step-rate** command. For example,

```
cpu0.set-step-rate "3/4"
```

will set the step rate of **cpu0** to 3/4; that is, three steps every four cycles.

If $q < p$, then some cycles will execute no step at all; if $q > p$, then some cycles will execute more than one step. The step rate parameters are currently limited to $1 \leqslant p \leqslant 128$ with $p = 2^k$ for some integer k, and $1 \leqslant q \leqslant 128$.

Setting a non-unity step rate can be used to better approximate the timing of a target machine averaging more or less than one instruction per cycle. It can also be used to compensate for Simics running instructions slower than actual hardware when it is desirable to have the simulated time match real time; specifying a lower step rate will cause simulated time go faster. Finally, a lower step rate may improve simulator performance by reducing the number of instructions executed between consecutive simulated timer interrupts. This is, however, to a high degree depending on the workload in question. Some workloads even benefit from a larger number of steps between timer interrupts.

The step rate is sometimes called IPC (instructions per cycle), and its inverse, the *cycle rate*, may be called CPI (cycles per instruction). The actual rates will depend on how many extra cycles are added by stalling.

Let us look at an example using a single Ebony card. We will first run 1 million steps with the default settings:

```
Wind River Simics 4.6 (build 4000 linux64) Copyright 2010-2011 Intel Corporation

Use of this software is subject to appropriate license.
Type 'copyright' for details on copyright and 'help' for on-line documentation.

simics> c 1000000
[cpu0] v:0xfff8a610 p:0x1fff8a610  mftbu r5
simics> ptime
processor                   steps             cycles    time [s]
cpu0                      1000000            1000000       0.010
```

The processor has run 1 million steps, taking 1 million cycles to execute them. Let us set the cycle rate to the value mentioned above, 3 steps for every 4 cycles:

```
simics> cpu0.set-step-rate "3/4"
simics> cb 1200000
simics> c
[cpu0] v:0xfff8a634 p:0x1fff8a634  bl 0xfff8a608
simics> ptime
processor                   steps             cycles    time [s]
```

```
cpu0                      1900000             2200000       0.022
simics>
```

When running the next 1.2 million cycles, the processor executes only 900000 steps, which corresponds to the 3/4 rate that we configured.

Suspending Time or Execution

It is possible to set the step rate to infinity, or equivalently, to suspend simulated time while executing steps. This is done by setting the *step_per_cycle_mode* processor attribute to one of the following values:

"constant"
: Steps are executed at the constant and finite rate specified in the *step_rate* attribute

"infinite"
: Steps are executed with no progress in simulated time

While time is suspended, the cycle counter does not advance, nor are any time events run. To the simulated machine this appears as if all instructions are run infinitely fast. Using the same example as above, we set the step per cycle mode to "infinite" to prevent the simulated time from advancing:

```
simics> cpu0->step_per_cycle_mode = "infinite"
simics> c 1000000
[cpu0] v:0xfff8a614 p:0x1fff8a614  cmpw r3,r5
simics> ptime
processor                   steps             cycles    time [s]
cpu0                      1000000                  0       0.000
simics>
```

Conversely, it is possible set the step rate to zero, thus suspending execution while letting simulated time pass. This can be done by stalling the processor for a finite time (see **Stalling** above) or by *disabling* the processor for an indefinite time. Disabling and re-enabling processors is done with the commands ⟨**processor**⟩**.enable** and ⟨**processor**⟩**.disable**.

The processor has executed 1 million steps but the simulated time has not advanced. Note that setting this mode would probably prevent a machine like Ebony from booting since many hardware events (like interrupts) are time-based.

Chapter 4

Multiprocessor Simulation

Simics can model systems with several processors, each with their own clock frequency. In this case the definition of how long a cycle is becomes processor-dependent. Ideally, Simics would make time progress and execute one cycle at a time, scheduling processors according to their frequency. However, perfect synchronization is exceedingly slow, so Simics *serializes* execution to improve performance.

Simics does this by dividing time into segments and serializing the execution of separate processors within a segment. The length of these segments is referred to as the *quantum* and is specified in seconds (this is similar to the way operating systems implement multitasking on a single-processor machine: each process is given access to the processor and runs for a certain time quantum). The processors are scheduled in a round-robin fashion, and when a particular processor *P* has finished its quantum, all other processors will finish their quanta before execution returns to *P*. The length of the time quantum can be set by using the command **cpu-switch-time**. The argument to **cpu-switch-time** is specified in either cycles referring to the first processor in the system or seconds.

As in the single-processor case, instruction execution and latency are defined with execution modes and timing interfaces. Simics does not define the order in which the processors are serialized, which means that if causality is to be preserved, processor-to-processor communications must have a minimum latency of one quantum. Another consequence of serializing the execution is that Simics will maintain strict sequential consistency. However, through careful use of the memory hierarchy interface, the user can choose to simulate other consistency models.

As an example, consider a dual-processor system where the first processor runs at 4 MHz and the second at 1 MHz. Setting **cpu-switch-time** to 10 will give a quantum of 2.5 simulated microseconds. During each quantum, the first processor will execute 10 steps, and the second 2 or 3 steps. During the following quanta the second processor will continue to execute 2 or 3 steps each quantum, but it is not defined exactly how many steps will be executed in each quantum. Stopping the simulation does not affect this schedule, so human interaction with Simics remains non-intrusive.

Note that if you are single-stepping (**step-instruction**) on a processor *P*, which has just executed the last cycle of a quantum, the next single-step will cause all other processors to advance an entire quantum and then *P* will stop after one step. This behavior makes it convenient to follow the execution of instructions on a particular processor. You can use the ⟨**processor**⟩**.ptime** command to see the current time on each particular processor in the simulated machine.

For a multi-processor simulation to run efficiently, the quantum should not be set too low, since a CPU switch causes simulator overhead. It should not be set below 10, and should preferably be set to 50 or higher. The default value is 1000. For a perfectly synchronized simulation, set the switch time to 1 (which will give a very slow simulation but is useful for detailed cache studies, for example). Note that all of the above remains essentially the same when running a distributed simulation (see next section).

Time events in Simics are executed when the processor on which they were posted run the triggering cycle during its quantum. However, it is possible to post *synchronizing* time events that will ensure that all processors have the same local time when the event is executed, independently of the time quantum. Synchronizing events can not be posted less than one time quantum in the future unless the simulation is already synchronized.

Let us have a look at a 2-machines setup containing two SPARC SunFire machine (with one processor each) to illustrate multiprocessor simulation. The processor in the first machine runs at 168MHz; the other runs at 56MHz (equal to 168/3). The time quantum (configured via the **cpu-switch-time** command) is 1000 cycles of the first processor, or 6 microseconds.

```
Wind River Simics 4.6 (build 4000 linux64) Copyright 2010-2011 Intel Corporation

Use of this software is subject to appropriate license.
Type 'copyright' for details on copyright and 'help' for on-line documentation.

simics> d1_cpu0->freq_mhz
168
simics> d2_cpu0->freq_mhz
56
simics> sim->cpu-switch-time
Current CPU switch time: 1000 cycles (0.000006 seconds)
simics> c 10000
[d1_cpu0] v:0xfffffffff0001364 p:0x1fff0001364  bne,pt %xcc, 0xfffffffff0001360
simics> ptime -all
processor                  steps              cycles    time [s]
d1_cpu0                    10000               10000       0.000
d2_cpu0                     3333                3333       0.000
```

While the first processor executed 10000 steps, the second processor completed 3333 steps, which corresponds to the ratio between the two frequencies (168MHz compared to 56MHz). Let us now examine the effects of the time quantum:

```
simics> c 30
[d1_cpu0] v:0xfffffffff0001364 p:0x1fff0001364  bne,pt %xcc, 0xfffffffff0001360
simics> ptime -all
processor                  steps              cycles    time [s]
d1_cpu0                    10030               10030       0.000
```

```
d2_cpu0                      3333                3333       0.000
```

Although the first processor ran 30 steps further, the second processor has not run the 10 steps that we would expect, and the frequency ratio is not respected anymore. This is the effect of the 1000 cycles time quantum: the first processor is scheduled for the next 1000 cycles and no other processor will be run until the quantum is finished. If we switch to the second processor and try to make it run one step further, we will observe the following:

```
simics> pselect d2_cpu0
simics> c 1
[d2_cpu0] v:0xfffffffff0001364 p:0x1fff0001364  bne,pt %xcc, 0xfffffffff0001360
simics> ptime -all
processor                   steps              cycles    time [s]
d1_cpu0                     11000               11000       0.000
d2_cpu0                      3334                3334       0.000
```

The second processor has run 1 step further as requested, but the first had to finish its time quantum before the second processor could be allowed to run, which explains its step count of 11000 compared to 10030 before. Let us now set the time quantum to 1:

```
simics> cpu-switch-time 1
The switch time will change to 1 cycles (for CPU-0) once all↩
 processors have synchronized.
simics> c 1
[d2_cpu0] v:0xfffffffff0001368 p:0x1fff0001368  nop
simics> ptime -all
processor                   steps              cycles    time [s]
d1_cpu0                     11000               11000       0.000
d2_cpu0                      3335                3335       0.000
```

Note that the new time quantum length will only become valid once all processors have finished their current time quantum. This is why stepping one more step forward with the second processor has not affected the first yet. Now let us select the first processor again, and run three steps:

```
simics> pselect d1_cpu0
simics> c 3
[d1_cpu0] v:0xfffffffff0001368 p:0x1fff0001368  nop
simics> ptime -all
processor                   steps              cycles    time [s]
d1_cpu0                     11003               11003       0.000
d2_cpu0                      3668                3668       0.000
simics> c 3
[d1_cpu0] v:0xfffffffff0001368 p:0x1fff0001368  nop
simics> ptime -all
```

```
processor                   steps              cycles    time [s]
d1_cpu0                     11006               11006       0.000
d2_cpu0                      3669                3669       0.000
simics>
```

All processors finished their 1000 cycles time quantum and started to run with the new 1 cycle value, which means that they are now advancing in lockstep. For every 3 steps performed by the first processor, the second executes 1.

Index