WIND RIVER

# Wind River® Simics® Accelerator

## USER'S GUIDE

## 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`

# Contents

# Chapter 1

# Introduction

This document describes the performance improvement features provided by Simics Accelerator. To get the best performance out of Simics you should also read the *Simulation Performance* chapter in the *Hindsight User's Guide* for general information about better Simics performance.

Simics Accelerator provides two major features: parallel simulation and page sharing. Parallel simulation allows you to use multithreading to run multi-machine simulations in parallel on multiprocessor hosts. You can also split the simulation and run it distributed across more than one host machine.

Page sharing finds identical memory pages and shares them to decrease memory consumption and increase execution performance. This is particularly useful when you have several machines running the same software.

# Chapter 2

# Multithreading

The easiest way to parallelize a simulation is to use Simics Accelerator's support for multithreading. It requires that the models used in the simulation are marked as *thread-safe*. The rest of this section describes how to use the multithreading capabilities of Simics.

With multithreading the simulation runs in a single Simics process: you control the entire simulation from a single point, and the entire simulation state gets saved in one checkpoint, just as when you run a single threaded simulation.

To use multithreading the configuration must be partitioned into *simulation cells*. Each cell contains a subset of the configuration objects in the simulation. The only communication allowed between cells is over *links*. A link transmits messages between objects with a *latency* measured in simulated time, for example an Ethernet cable.

Dividing the system into cells can be done automatically via the Simics' component system. This makes it easy to parallelize an existing model.

## 2.1 Multithread-Ready Models

Most models provided with Simics can run multithreaded and are thus marked *thread-safe*. Loading modules that are not marked *thread-safe* will result in a warning message and multithreading will be disabled. Please contact your Wind River representative if you are running a model that is not multithread-ready and you want to utilize multithreading.

If you developed your own models of devices, you should refer to the *Model Builder User's Guide* to learn how to make them multithread-compatible.

Whenever possible, all default components provided with Simics create *simulation cells* that can be multithreaded. For example, instantiating two *MPC8641-Simple* boards in the same Simics session will create two *cells*, which can be scheduled on two simulation threads. The maximum possible parallelism is limited by the number of cells in a session (as well as the number of processor cores on your host, of course). You can list the cells instantiated in a configuration with the following command:

```
simics> list-objects -all type = cell
```

## 2.2 Enabling and Disabling Multithreading

Multithreading is enabled by default. It can be turned off using the command

```
simics> disable-multithreading
```

and on again with

```
simics> enable-multithreading
```

This command will also check that the configuration looks reasonable before switching on multithreading, and warn you if something is incorrect.

## 2.3 Controlling Thread Synchronization

To allow multithreaded simulation to perform well, Simics lets each thread run for a certain amount of virtual time on its own before it needs to resynchronize with the other threads. This time span is the synchronization latency. Because of the synchronization latency, Simics does not allow communication between objects of different cells. Even if all accesses were properly locked and performed in a thread-safe way, the objects would have no way to control at what time their access would be done in the other cell, and the simulation would stop being deterministic.

The solution is to communicate via *link objects*. Link objects ensures that messages send from one cell are delivered at the expected virtual time in the other cell, at the cost of a virtual time delay in the transmission. For links to send messages deterministically, the delay in transmission must be greater or equal to the synchronization latency. For this reason, the synchronization latency is often called the minimum latency for link communication.

The next two sections explain how to control the synchronization latency—and the link latencies—in multithreaded simulations.

### The Simple Way

By default, Simics creates a single synchronization domain called **default_sync_domain**. Cells created later in the simulation will be attached to this synchronization domain, unless specified otherwise. Thus the synchronization latency in the simulation will be controlled by the *min-latency* attribute set in **default_sync_domain**.

Note that for the time being, the latency of a synchronization domain can not be changed after the domain has been created. We hope to remove that limitation in the future.

The simplest way to control the synchronization latency is to use the **set-min-latency** command, which will immediately create a default synchronization domain with the required latency. Because it relies on creating a synchronization domain, **set-min-latency** can only be used before the simulation has been configured, and only once.

```
simics> set-min-latency 0.01
simics> list-objects type = sync_domain
Component Class  Object
-----------------------

Class          Object
```

```
-----------------------------------
<sync_domain>  default_sync_domain

simics> default_sync_domain->min_latency
0.01
```

One important thing to remember is that the time quantum in each multiprocessor cell must be less than half the minimum latency. In other words: sync_latency $>$ 2 $\times$ time_quantum for every multiprocessor cell in the system. Simics will print an error if this condition is not respected.

### Understanding Synchronization Domains

Synchronization latencies can be controlled in a much finer way. Synchronization domains can be organized in an hierarchy that allows different cells to be synchronized with different latencies. This organization is the foundation of the new domain-based distribution system, described in the next section.

Let us build a networked system with two-tightly coupled machines communicating on a very fast network, associated with a control server that sends a command from time to time. The two machines require a low communication latency, while the communication latency between them and the server does not matter. Using a hierarchy of two domains allows all latency requirements to be fulfilled without sacrificing performance:

```
Top-domain (latency 1.0s)
 -> Server cell
 -> Sub-domain (latency 1e-6s)
     -> Machine0 cell
     -> Machine1 cell
```

In that configuration, the two machines can communicate with a latency of `1e-6s` while the communication latency between the machines and the server is `1s`. In practice, this allows Simics to give the server a `1s` synchronization window with the two machines, hence much less synchronization overhead and a better usage of parallel simulation.

More concretely, in Simics, the domains are setup in the following way (in Python):

```
domains = [OBJECT("top_domain", "sync_domain",
                  min_latency = 1.0),
           OBJECT("sub_domain", "sync_domain",
                  domain = OBJ("top_domain"),
                  min_latency = 1e-6)]
SIM_set_configuration(domains)
```

Cells created automatically can be assigned to a domain by using the *domain* attribute of the corresponding top-component. It is also possible to set a cell's *sync_domain* attribute when creating it manually.

### Setting Latencies: the Complete Rules

Latencies must obey certain rules for the domain hierarchy to work properly:

- The time quantum of a multiprocessor cell must be less than half the latency of the domain that contains the cell. The reason for this restriction is that the synchronization system considers a cell as a single unit and does not cope with the fact that the processors inside the cell are scheduled in a round-robin fashion. Simics check this requirement and prints an error message if a domain latency is incompatible with a cell time quantum.
- The latency of a child domain must be less than half the latency of its parent domain. This restriction is once again related to how synchronization events are scheduled. Simics check for this requirement and adjust the latency of the child domain automatically while printing a warning.
- The latency of a domain must be greater than the length of two cycles of the slowest processor it contains. Simics uses cycles as lowest time unit for posting events, so synchronization can not be ensured if the latency resolution is too small. Simics checks this requirement and prints an error message if a domain latency is incompatible with one of the processor.
- A link may not have a latency smaller than the one of the lowest domain in the hierarchy that contains the cells the link is connected in. In other words: the link must obey the highest latency between the systems it is connected to. Simics checks this requirement and adjust the link latency automatically upward if necessary while printing a warning.
- Once set, latencies may not be changed. This is a limitation in Simics that we hope to remove in future versions.

## 2.4 Multithreading and Scripting

Commands and script branches are never run multithreaded, thus parallelism can be safely ignored most of the time when scripting Simics. However, multithreading a simulation has side-effects that may cause scripts to behave in a correct but indeterministic way. If we consider the following script, in a configuration consisting of two cells, **cell0** and **cell1**:

```
cell0_console.break "foo"
c
cell1_console.input "bar"
```

Even with **cell0** and **cell1** running in parallel, the simulation will stop properly when the text breakpoint in **cell0** is triggered. However, **cell1** is not at a deterministic point in time: the only thing known about it is that it is within a certain window of virtual time in which it is allowed to drift without needing to re-synchronize with **cell0**, as explained in the previous section. So running this script twice in a row may not produce exactly the same results.

In many cases, it does not matter and the scripts will work fine. If perfect determinism is required, it is possible to save a checkpoint and run the sensitive part of the simulation single-threaded.

One aspect of multithreading that affects scripting directly is Python scripting. Hap handlers are run in the thread where they are triggered, which means that the same handler can run in parallel on different host processors. If the handler uses global state, it must use proper locks to access it. In general, this is not a problem since most haps are triggered for a specific object, so their handlers will only run in the thread where this object is scheduled. Some haps are triggered globally, however, and care must be taken when responding to them.

Python scripts are run with the global Python lock taken, so Python scripts never *really* run in parallel. However, the Python interpreter will schedule Python threads as it sees fit, so Python code that may run in several threads (device or extension code, hap handlers) should not assume that it has full control of the execution. The Python lock is also released every time a Simics API functions is called (including implicit calls like reading an attribute value).

When running Python scripts in a simulation thread, the script should not to access state that is in a different cell, since this cell might be running on another host processor. When in need to access the whole simulation state, a callback function can be scheduled with ***SIM_run_alone()*** (this is currently how script branches and commands are handled).

Finally, running commands in the simulation thread is *not* allowed, as the CLI parser is not thread-safe and might cause unexpected problems. Commands must be scheduled with ***SIM_run_alone()***. It is also possible to rewrite scripts to access directly objects and attributes instead of using the commands directly.

## 2.5 Dynamic Load Balancing

Simics uses dynamic load balancing to distribute the simulation workload across the available hardware resources (host CPUs). The dynamic load balancer ensures that all *simulation cells* are simulated as efficiently as possible, given the available number of host cores.

Using as many cells as possible can potentially improve performance since this increases the parallelism of the simulation. Having many cells also makes it easier for the dynamic load balancer to keep all threads fully loaded.

By default, Simics spawns at most as many threads as there are host cores, but it is possible to limit this number using the **set-thread-limit** command. Setting a thread limit may be useful if the physical machine is shared by multiple users.

# Chapter 3

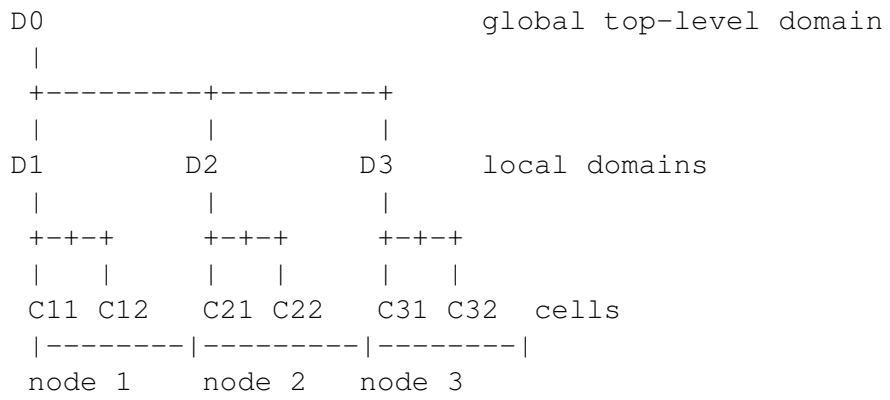
# Distributed Simulation

Distributed simulation is used for connecting multiple Simics processes, possibly running on different host machines, to run synchronously and exchange data in a reliable and deterministic way. It replaces the old *Simics Central* mechanism, which does not support multithreaded simulation processes.

## 3.1 Configuration

The Simics processes taking part in the combined simulation, here called *nodes*, are configured and managed individually. Each node will set up and run its own configuration and have its own name space. It will be controlled by its own command line or graphical interface.

Nodes are strung together by letting the local top-level synchronization domain in one node have a domain in another node as parent. Typically, there will be one global top-level domain in one node controlling the domains in all other nodes:




In the above diagram, **D0-D3** are synchronization domains and **C11-C32** cells. **D1**, **C11** and **C12** are all in node 1, and so on. The top-level domain **D0** could be placed either in a node of its own, without any cells, or in one of the other nodes. We will here assume it is located in node 1, the *server node*; the other nodes are then *clients*.

Domains in different nodes connect by proxies, which themselves connect over the network. The relation between D0 and D3 above is set up like follows:

```
          /     D0           sync_domain
   node 1 |     |
          \     D3_proxy     remote_sync_node
                :
             (network connection)
                :
          /     D0_proxy     remote_sync_domain
   node 3 |     |
          \     D3           sync_domain
```

The **remote_sync_domain** in the client node, **D0_proxy**, is created explicitly in the configuration for that node. The **remote_sync_node** in the server node is created automatically by a special server object when **D0_proxy** connects to the server node.

When a node has finished its configuration, it must inform the server to allow other clients to connect. This is done by setting to `None` the *finished* attribute of the **remote_sync_domain** object, or the **remote_sync_server** in the server. As a result, node configuration is done in sequence.

The default domain used by cells is **default_sync_domain**, so by using this as the local domain name, existing non-distributed configurations can be re-used. It is also a good idea to use the same name for the **remote_sync_domain** as for the actual top-level domain it is a proxy for. That way, it will matter less in what node the top-level domain is placed.

The configuration script for a single node could look like the following Python fragment:

```
srv_host = "serverhost"  # machine the server node runs on
srv_port = 4567          # TCP port the server listens on

# Start by creating the global and/or local domain objects:

if this_is_the_server_node:
    topdom = SIM_create_object("sync_domain", "top_domain",
                               [["min_latency", 0.04]])
    rss = SIM_create_object("remote_sync_server", "rss",
                            [["domain", topdom], ["port", srv_port]])
else:
    # Client nodes: create a proxy for the top-level domain.
    # This will initiate a connection to the server.
    topdom = SIM_create_object("remote_sync_domain", "top_domain",
                               [["server",
                                 "%s:%d" % (srv_host, srv_port)]])
# create a local domain to be a parent for the cells in this node
SIM_create_object("sync_domain", "default_sync_domain",
                  [["sync_domain", topdom], ["min_latency", 0.01]])

# --- Here the rest of the node should be configured. ---
```

```
if this_is_the_server_node:
    rss.finished = None     # let clients connect to the server
else:
    topdom.finished = None  # let other clients connect to the server
```

At the end of this script, the configuration is finished for that node. Note that other nodes may not have finished theirs yet—the simulation cannot start until the entire system has been set up. The user can just wait for this to happen, or write a mechanism to block until the system is ready; see the section about global messages below.

## 3.2 Links

Links work across nodes in the same way as in a single process simulation. Using the same global ID for links in two different nodes ensures that they are considered as the same link in the distributed simulation. The global ID for a link is set using the *global_id* attribute when the link is created.

There is one important aspect of link distribution that should be taken into account when creating distributed configuration.

When creating single-session configuration, Simics provides only one object namespace, which means that all objects have a unique name in that session. This property is used to keep link message delivery deterministic when no other way of comparing the messages is available. To be more precise, messages arriving from different senders to the same receiver at the same cycle are sorted according to the pair (sender name, sender port).

In distributed sessions however, Simics does not impose a single object namespace. This allows several objects with the same name to be connected to the same distributed link. As a consequence, the delivery of messages as described in the previous paragraph may become indeterministic again, since different sender may report the same (sender name, sender port) pair. Distributed links report an error if such a configuration is found.

The solution is to name differently the various boards or machines that compose the complete distribution configuration.

---

**Note:** Deleting a distributed link is not supported.

---

## 3.3 Global Messages

There is a supporting mechanism for sending simple messages to all nodes in the combined system: ***VT_global_message(msg)*** will trigger the hap `Core_Global_Message` with that string as parameter in each node, including the one sending the message. A listener of the message could look like:

```
def global_message_callback(obj, idx, msg):
    print "got message", msg
SIM_hap_add_callback("Core_Global_Message",
```

```
global_message_callback, None)
```

Global messages will arrive and be processed during a call to ***SIM_process_work()***. This is useful for blocking further execution of a script until a certain message has arrived.

Global messages can be used to assist in configuration and running a distributed system. Possible uses include:

- waiting for all nodes to finish their configuration
- starting and stopping the simulation on all nodes as the same time
- broadcasting commands to all nodes
- sending data to a single requesting node
- saving the configuration of all nodes

This facility is *not intended for simulation use*; message delivery is reliable but not deterministic in timing. It should be regarded as a low-level mechanism to be used as a building block for higher-level communication.

## 3.4 Running the simulation

Each node will have to start the simulation for the combined system to make progress. If one node stops the simulation—by user intervention or because of a coded breakpoint—the remaining nodes will be unable to continue, because the top-level synchronization domain keeps cells in different nodes from diverging.

Each node can only access the objects it has defined locally. This means that inspection and interaction with the system must be done separately for each node. A combined front-end interface is not available for Simics at this time.

When one Simics process terminates, the other nodes will automatically exit as well.

## 3.5 Saving and restoring checkpoints

The state of a distributed simulation can be saved by issuing **write-configuration** in each node separately. To restore such a checkpoint, start an equal number of (empty) Simics processes and read the saved configuration for each node.

Note that the server host name and port number are saved in the configuration files. These have to be edited if the checkpoint is restored with a different server host, or if the port number needs to be changed.

Note as well that the configurations must be read in sequence again, using the *finished* attribute to control which session is allowed to connect. However, the order of read-configuration sequence does not matter, as long as the server is started first.

## 3.6 Security

The distributed simulation facility has no authentication mechanism and is not security-hardened. It should therefore only be used in a trusted network environment.

# Chapter 4

# Page-Sharing

When running multiple instances of the same target system, in a virtual network or similar, it is likely that many of the RAM, ROM, flash, or disk pages in each system are identical to the others. Simics Accelerator adds a new *page-sharing* feature which takes advantage of this fact to reduce host memory consumption and increase execution performance.

When the page-sharing feature is activated, at certain trigger points Simics will examine the contents of a page, comparing it with other pages examined earlier. If an identical match is found, the page is removed and instead set to share data with the other page(s).

When many pages are shared the host memory that Simics uses will be reduced and consequently it will take longer until the memory-limit is reached. If the frequency of reaching Simics memory-limit is a factor limiting performance, execution performance will also increase. Simics can also take advantage of the fact that the page is shared by sharing the generated JIT code corresponding to a shared page. This can lead to improved performance since the JIT code only needs to be created once, but also thanks to better instruction caching when many target processors use the same JIT code.

Shared pages are always read-only, if a shared page is written to, it will automatically be un-shared and the writer will be given a private copy of the page. Consequently, page-sharing works best for pages containing only instructions or data pages which are never or rarely modified.

To activate page-sharing, use the command **enable-page-sharing**. This command also has a *-now* command argument which causes all pages to be analyzed and possibly shared directly.

To monitor how much memory that is saved due to the page-sharing feature, the **system-perfmeter** has a *-shared* switch which shows how much memory in total that has been saved, in each measurement sample. (See chapter *Simulation Performance* in the *Hindsight User's Guide*.)

Use **disable-page-sharing** to deactivate the page-sharing feature. This will not cause already shared pages to be un-shared but no more pages will be shared.

# Index