virtutech

Simics/Niagara Simple Target Guide

Simics Version	3.0
Revision	1376
Date	2007-01-24

VIRTUTECH CONFIDENTIAL

Contents

Chapter 1

About Simics Documentation

1.1 Conventions

Let us take a quick look at the conventions used throughout the Simics documentation. Scripts, screen dumps and code fragments are presented in a `monospace` font. In screen dumps, user input is always presented in bold font, as in:

```
Welcome to the Simics prompt
simics> this is something that you should type
```

Sometimes, artificial line breaks may be introduced to prevent the text from being too wide. When such a break occurs, it is indicated by a small arrow pointing down, showing that the interrupted text continues on the next line:

```
This is an artificial ↵
line break that shouldn't be there.
```

The directory where Simics is installed is referred to as `[simics]`, for example when mentioning the `[simics]/README` file. In the same way, the shortcut `[workspace]` is used to point at the user's workspace directory.

1.2 Simics Guides and Manuals

Simics comes with several guides and manuals, which will be briefly described here. All documentation can be found in `[simics]/doc` as Windows Help files (on Windows), HTML files (on Unix) and PDF files (on both platforms). The new Eclipse-based interface also includes Simics documentation in its own help system.

Simics Installation Guide for Unix and for Windows

These guides describe how to install Simics and provide a short description of an installed Simics package. They also cover the additional steps needed for certain features of Simics to work (connection to real network, building new Simics modules, . . .).

Simics User Guide for Unix and for Windows

These guides focus on getting a new user up to speed with Simics, providing information on Simics features such as debugging, profiling, networks, machine configuration and scripting.

Simics Eclipse User Guide

This is an alternative User Guide describing Simics and its new Eclipse-based graphical user interface.

Simics Target Guides

These guides provide more specific information on the different architectures simulated by Simics and the example machines that are provided. They explain how the machine configurations are built and how they can be changed, as well as how to install new operating systems. They also list potential limitations of the models.

Simics Programming Guide

This guide explains how to extend Simics by creating new devices and new commands. It gives a broad overview of how to work with modules and how to develop new classes and objects that fit in the Simics environment. It is only available when the DML add-on package has been installed.

DML Tutorial

This tutorial will give you a gentle and practical introduction to the Device Modeling Language (DML), guiding you through the creation of a simple device. It is only available when the DML add-on package has been installed.

DML Reference Manual

This manual provides a complete reference of DML used for developing new devices with Simics. It is only available when the DML add-on package has been installed.

Simics Reference Manual

This manual provides complete information on all commands, modules, classes and haps implemented by Simics as well as the functions and data types defined in the Simics API.

Simics Micro-Architectural Interface

This guide describes the cycle-accurate extensions of Simics (Micro-Architecture Interface or MAI) and provides information on how to write your own processor timing models. It is only available when the DML add-on package has been installed.

RELEASENOTES and LIMITATIONS files

These files are located in Simics's main directory (i.e., `[simics]`). They list limitations, changes and improvements on a per-version basis. They are the best source of information on new functionalities and specific bug fixes.

Simics Technical FAQ

This document is available on the Virtutech website at `http://www.simics.net/support`. It answers many questions that come up regularly on the support forums.

Simics Support Forum

The Simics Support Forum is the main support tool for Simics. You can access it at `http://www.simics.net`.

Other Interesting Documents

Simics uses Python as its main script language. A Python tutorial is available at `http://www.python.org/doc/2.4/tut/tut.html`. The complete Python documentation is located at `http://www.python.org/doc/2.4/`.

Chapter 2

Simics/Niagara Overview

2.1 Introduction

Simics/Niagara Simple models the processor and memory of a Sun Fire T2000 server with an UltraSPARC T1 processor. The processor is running at 5MHz and there are 256 MB of memory. A single disk image with Solaris is also supported.

Virtutech does not provide any disk images with Solaris for Niagara Simple, due to licensing issues. No OpenBoot PROM (OBP) image is not provided either. These files can be downloaded from The OpenSPARC web-site.

2.2 Supported Hardware

The niagara-simple has only support for the SUN4V console and RTC device. There are no other devices modelled.

A good guide to the Sun Fire T2000 servers can be found in the Sun System Handbook, available online at: `http://sunsolve.sun.com/handbook_pub/`

2.3 Required Files

Some files required to run the Niagara Simple system are not included in the Simics distribution. These files can be found on the opensparc.org web site, bundled with the UltraSPARC T1 simulator from Sun.

Copy the files in the `S10image/` directory of the Sun simulator package to `[simics]/targets/niagara-simple/images/`. The files needed are:

- 1up-hv.bin
- 1up-md.bin
- 1g2p-hv.bin
- 1g2p-md.bin
- 1g32p-hv.bin

- 1g32p-md.bin
- disk.s10hw2
- openboot.bin
- q.bin
- reset.bin

Chapter 3

Simulated Machines

Simics scripts for starting Niagara Simple machines are located in the `[workspace]/targets/niagara-simple/` directory, while the actual configuration scripts can be found in `[simics]/targets/niagara-simple/`.

3.1 niagara-simple-solaris

The niagara-simple machine is the only configuration distributed. It has an UltraSPARC T1 processor running at 5 MHz, and 256 MB of memory. One, two or all 32 strands (virtual processors/hardware threads) can be enabled. The default configuration can be modified as described in section 3.2. The reason that the memory size, processor frequency and number of strands cannot be changed is that they are hardcoded in binary files used by the OBP.

3.1.1 Niagara-simple Scripts

`niagara-simple-solaris-common.simics`
Starts the Niagara-simple machine with the default configuration.

3.2 Parameters for Machine Scripts

The following parameters can be set before running the `niagara-simple-solaris-common.simics`. Other `.simics` scripts may set some of the parameters unconditionally, and do not allow the user to override them.

3.2.1 niagara-simple-solaris-common

$do_boot
Set to `no` to stop at OBP prompt, without booting the OS.

$do_login
Set to `no` to prevent the script from logging in as root automatically when the operating system has reached the login prompt.

$freq_mhz
The clock frequency in MHz of the processors. This value can currently not be changed from 5 MHz.

$rtc_time
Date and time of the real-time clock at boot.

$num_cpus
The number of strands (virtual processors) in the machine. The supported values are currently 1, 2 and 32.

$use_simicsfs
when set to `yes` (default), a "diff" disk image is added to the default disk image files from Sun, providing SimicsFS support.

Chapter 4

Supported Components

The following sections list components that are supported for the Niagara Simple architecture. There also exist other components in Simics, such as various PCI devices, that may work for Niagara Simple but that have not been tested.

The default machines are constructed from components in the `-system.include` files in `[simics]/targets/niagara-simple/`. See the Configuration and Checkpointing chapter in the Simics User Guide for information on how to define your own machine, or make modifications to an existing machine.

4.1 Niagara Simple Components

4.1.1 niagara-simple-system

Description

The "niagara-simple-system" component represents a fake Sun Fire T2000 server with an UltraSPARC T1 processor memory and a single disk image, but with no other devices.It is currently not possible to change the configuration, except for the number of processors.

Attributes

cpu_frequency

Required attribute; **read/write** access; type: **Integer**.

Processor frequency in MHz.

num_cores

Required attribute; **read/write** access; type: **Integer**.

Number of processor cores in the system. 1 or 8.

rtc_time

Required attribute; **read/write** access; type: **String**.

The date and time of the Real-Time clock.

strands_per_core

Required attribute; **read/write** access; type: **Integer**.

Number of active processor strands per core in the system. 1, 2 or 4. There are always 4 strands created.

Commands

create-niagara-simple-system [*"name"*] *cpu_frequency num_cores strands_per_core "rtc_time"*

Creates a non-instantiated component of the class "niagara-simple-system". If *name* is not specified, the component will get a class-specific default name. The other arguments correspond to class attributes.

new-niagara-simple-system [*"name"*] *cpu_frequency num_cores strands_per_core "rtc_time"*

Creates an instantiated component of the class "niagara-simple-system". If *name* is not specified, the component will get a class-specific default name. The other arguments correspond to class attributes.

<**niagara-simple-system**>**.get-prom-env [*"variable"*]**
Prints an OBP variable with its value, or all variables if no argument is specified. Only variables with string, integer, boolean and enumeration types are supported.

<**niagara-simple-system**>**.info**
Print detailed information about the configuration of the device.

<**niagara-simple-system**>**.set-prom-defaults**
Restores all OBP variables to their default values.

<**niagara-simple-system**>**.set-prom-env *"variable"* (*int*|*"string"*)**
Sets the value of an OBP variable in the NVRAM. Only variables with string, integer, boolean and enumeration types are supported.

<**niagara-simple-system**>**.status**
Print detailed information about the current status of the device.

Connectors

Name	**Type**	**Direction**
com[1-2]	serial	down

4.2 Standard Components

4.2.1 std-serial-link

Description

The "std-serial-link" component represents a standard Serial link.

Commands

create-std-serial-link [*"name"*]

Creates a non-instantiated component of the class "std-serial-link". If *name* is not specified, the component will get a class-specific default name. The other arguments correspond to class attributes.

new-std-serial-link [*"name"*]

Creates an instantiated component of the class "std-serial-link". If *name* is not specified, the component will get a class-specific default name. The other arguments correspond to class attributes.

<**std-serial-link**>**.info**

Print detailed information about the configuration of the device.

<**std-serial-link**>**.status**

Print detailed information about the current status of the device.

Connectors

Name	**Type**	**Direction**
serial[0-1]	serial	any

4.2.2 std-text-console

Description

The "std-text-console" component represents a serial text console.

Attributes

bg_color

Optional attribute; **read/write** access; type: **String**.

The background color.

fg_color

Optional attribute; **read/write** access; type: **String**.

The foreground color.

height
Optional attribute; **read/write** access; type: **Integer**.
The height of the console window.

title
Optional attribute; **read/write** access; type: **String**.
The Window title.

width
Optional attribute; **read/write** access; type: **Integer**.
The width of the console window.

win32_font
Optional attribute; **read/write** access; type: **String**.
Font to use in the console on Windows host.

x11_font
Optional attribute; **read/write** access; type: **String**.
Font to use in the console when using X11 (Linux/Solaris host).

Commands

**create-std-text-console [*"name"*] [*"title"*] [*"bg_color"*] [*"fg_color"*] [*"x11_font"*] [*"win32_font"*] [*w*

Creates a non-instantiated component of the class "std-text-console". If *name* is not specified, the component will get a class-specific default name. The other arguments correspond to class attributes.

**new-std-text-console [*"name"*] [*"title"*] [*"bg_color"*] [*"fg_color"*] [*"x11_font"*] [*"win32_font"*] [*wi*

Creates an instantiated component of the class "std-text-console". If *name* is not specified, the component will get a class-specific default name. The other arguments correspond to class attributes.

<**std-text-console**>**.info**
Print detailed information about the configuration of the device.

<**std-text-console**>**.status**
Print detailed information about the current status of the device.

Connectors

Name	**Type**	**Direction**
serial	serial	up

4.2.3 std-server-console

Description

The "std-server-console" component represents a serial console accessible from the host using telnet.

Attributes

telnet_port

Required attribute; **read/write** access; type: **Integer**.

TCP/IP port to connect the telnet service of the console to.

Commands

create-std-server-console [*"name"*] *telnet_port*

Creates a non-instantiated component of the class "std-server-console". If *name* is not specified, the component will get a class-specific default name. The other arguments correspond to class attributes.

new-std-server-console [*"name"*] *telnet_port*

Creates an instantiated component of the class "std-server-console". If *name* is not specified, the component will get a class-specific default name. The other arguments correspond to class attributes.

<**std-server-console**>**.info**

Print detailed information about the configuration of the device.

<**std-server-console**>**.status**

Print detailed information about the current status of the device.

Connectors

Name	Type	Direction
serial	serial	up

4.3 Base Components

The base components are abstract classes that contain generic component attributes and commands available for all components.

4.3.1 component

Description

Base component class, should not be instantiated.

Attributes

connections

Optional attribute; **read/write** access; type: **[[sos]**$*$**]**.

List of connections for the component. The format is a list of lists, each containing the name of the connector, the connected component, and the name of the connector on the other component.

connectors

Pseudo class attribute; **read-only** access; type: **D**.

Dictionary of dictionaries with connectors defined by this component class, indexed by name. Each connector contains the name of the connector "type", a "direction" ("up", "down" or "any"), a flag indicating if the connector can be "empty", another flag that is set if the connector is "hotplug" capable, and finally a flag that is TRUE if muliple connections to this connector is allowed.

instantiated

Optional attribute; **read/write** access; type: **b**.

Set to TRUE if the component has been instantiated.

object_list

Optional attribute; **read/write** access; type: **D**.

Dictionary with objects that the component consists of.

object_prefix

Optional attribute; **read/write** access; type: **String**.

Object prefix string used by the component. The prefix is typically set by the **set-component-prefix** command before the component is created.

top_component

Optional attribute; **read/write** access; type: **Object**.

The top level component. Attribute is not valid until the component has been instantiated.

top_level

Optional attribute; **read/write** access; type: **b**.

Set to TRUE for top-level components, i.e. the root of a hierarchy.

4.3.2 top-component

Description

Base top-level component class, should not be instantiated.

Attributes

components

Optional attribute; **read/write** access; type: **[o∗]**.

List of components below the the top-level component. This attribute is not valid until the object has been instantiated.

cpu_list

Optional attribute; **read/write** access; type: **[o∗]**.

List of all processors below the the top-level component. This attribute is not valid until the object has been instantiated.

Chapter 5

Miscellaneous Notes

5.1 Notes on Solaris for Niagara

- For information about system administration of Solaris, see the `http://docs.sun.com` web site.

5.2 Changing the Processor Clock Frequency

The clock frequency of a simulated processor can be set arbitrarily in Simics. This will not affect the actual speed of simulation, but it will affect the number of instructions that need to be executed for a certain amount of simulated time to pass. If your execution only depends on executing a certain number of instructions, increasing the clock frequency will take the same amount of host time (but a shorter amount of target time). However, if there are time based delays of some kind in the simulation, these will take longer to execute.

At a simulated 1 MHz, one million target instructions will correspond to a simulated second (assuming the simple default timing of one cycle per instruction). At 100 MHz, on the other hand, it will take 100 million target instructions to complete a simulated second. So with a higher clock frequency, less simulated target time is going to pass for a certain period of host execution time.

If Simics is used to emulate an interactive system (especially one with a graphical user interface) it is a good idea to set the clock frequency quite low. Keyboard and mouse inputs events are handled by periodic interrupts in most operating systems, using a higher clock frequency will result in longer delays between invocations of periodic interrupts. Thus, the simulated system will feel slower in its user response, and update the mouse cursor position etc. less frequently. If this is a problem, the best technique for running experiments at a high clock frequency is to first complete the configuration of the machine using a low clock frequency. Save all configuration changes to a disk diff (like when installing operating systems). Then change the configuration to use a higher a clock frequency and reboot the target machine.

Note that for a lightly-loaded machine (for example, working at an interactive prompt on a serial console to an embedded Linux system), Simics will often execute quickly enough at the real target clock frequency that there is no need to artifically lower it.

Chapter 6

Limitations

6.1 Limitations of the Simulated Model

- Not all error registers in the processor are implemented.
- Performance control registers, and counters are not implemented
- Modular Arithmetic is not implemented.
- Changing the processor frequency nad memory size is not yet supported.

6.2 Other Limitations

- The Solaris version of SimicsFS does not support truncating files.

Index

Virtutech, Inc.

1740 Technology Dr., suite 460
San Jose, CA 95110
USA

Phone +1 408-392-9150
Fax +1 408-608-0430

http://www.virtutech.com