WIND RIVER

Wind River® Simics® Hindsight

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

Part I

Introduction

Chapter 1

Introduction

This document describes the features of Simics Hindsight, with in-depth explanations on their usage and implementation. A gentler introduction is provided in the *Getting Started* document.

The first chapters cover various aspects of Simics Hindsight, from its command-line and scripting to the way simulation can be inspected and manipulated. The second part of the manual is dedicated to software debugging, both with Simics Hindsight stand-alone or when using it as a back-end for other debuggers such as Wind River Workbench or GDB.

Simics Hindsight includes a simple simulation-only hardware model, the Quick Start Platform (QSP). This model can be used to get started with Simics, and it can be used to run user level application software that only interfaces to the operating system APIs. Its devices are functionally complete for their intended purpose. Please find more information in the QSP Target Guides.

Chapter 2

Simulation Concepts

2.1 The Limits of Simulation

Simics is a system-level instruction set simulator. This means that:

- Simics models the target system at the level of individual instructions, executing them one at a time.
- Simics models the binary interface to hardware in sufficient detail that any software that runs on the real hardware will also run on Simics.

In practice, what this means is that there is no code that is too "low-level"—Simics can run, and debug, any kind of software: firmware, hardware drivers, operating systems, user-level applications, whatever. There are some caveats, though:

- Simics's model of time is rather simple; for example, it assumes by default that all instructions take the same amount of time to run. It is not difficult to write a program that uses this fact to detect the difference between Simics and real hardware. However, this is seldom a problem with real-world software.

  You can read more about Simics and time in the *Understanding Simics Timing* application note.

- The hardware models must be detailed enough. Models of nontrivial pieces of hardware do not typically model all functions and details of that hardware, so it is possible to write a program that detects the difference between Simics and real hardware by probing unimplemented functions. However, any given piece of software can be accommodated by extending the hardware models to cover the missing functions.

  You can read more about hardware modeling in the *Simics Model Builder User's Guide*.

2.2 Non-intrusive Inspection and Debugging

Simics has powerful built-in inspection and debugging facilities. These include:

- Inspecting registers, memory, and hardware state.

- Modifying register and memory contents, and hardware state.
- Setting (and triggering) breakpoints and watchpoints.
- Powerful scripting support for all of this.

Because these are implemented in the simulator, no debugging software needs to be on the target at all. As a result, the debugging machinery is completely invisible to the target (and thus to any software running on it).

2.3 Simulated Time

One of the most powerful properties of full-system simulation is that time inside the simulation and time in the real world are two completely different things. This brings a number of substantial benefits:

- You can pause the simulation at any time, and the software running in the simulated simply *cannot* detect this. This allows you to inspect (and optionally modify) the state even at points where real hardware would be unable to stop.
- You can save the state of the simulation to disk (this is called a *checkpoint*), and start again from that point at any time, any number of times.
- The simulation is completely deterministic. Every time you start from the same state (such as a checkpoint), the *exact* same thing will happen. This can be tremendously useful when hunting down certain types of bugs.
- The simulation can be reversed, making it possible to answer questions such as "This value is garbage now. When did this happen, and who did it?"; simply set a write breakpoint and run backward until it triggers, and you have your culprit.

These advantages apply to the entire simulated system, whether it is a single target machine or an entire network.

Part II

Feature Overview

Chapter 3

Using Simics from the Command Line

This chapter describes command line options accepted by Simics and the environment variables Simics recognizes.

Even on Windows host, running Simics from the command line is useful, for example, when running automatic test scripts, profiling or benchmarking. Whole test sessions can be automated by using CLI or Python and executed directly from Window's command shell.

Below is an example on how Simics can be started from the command line using a Simics workspace.

Windows

Run the `simics.bat` script in your workspace directory.

```
[workspace]> simics.bat targets\mpc8641-simple\mpc8641d-simple-linux.simics
```

Unix

Run the `./simics` script in your workspace directory.

```
joe@computer:[workspace]$ ./simics targets/mpc8641-simple/mpc8641d-simple-linux.simics
```

Simics can also be started by running a similar start script in the `[simics]\bin` directory (`[simics]/bin` on Unix). This will run Simics without a workspace, and is usually not recommended.

This will run the `mpc8641d-simple-linux.simics` script, which will start the *MPC8641-Simple* machine with its default configuration.

To quit the Simics shell you can type `quit` at the Simics prompt.

3.1 Simulation Modes

The *simulation mode* (or *execution mode*) is used to control the availability of certain features: instruction and data profiling, memory timing. To allow for maximum performance when

these features are not used, Simics comes with separate processor implementations. By default, Simics uses the fastest possible mode ("normal").

Simics started from a command line:

To select a simulation mode, the corresponding command line flag (`-stall` or `-fast`) is passed to Simics. For example, to select "Stall mode":

Windows

```
$ cd [workspace]
$ simics.bat -stall ...
```

Unix

```
$ cd [workspace]
$ ./simics -stall ...
```

Simics does not support switching processor implementations at runtime. To change from one mode to another, use Simics's checkpoint feature (see chapter 6 for full details) to save the simulation state and restart Simics in the new mode from the checkpoint:

```
simics> write-configuration at-workload-start.ckpt
simics> quit
```

and then start from the new checkpoint in the new simulation mode:

Windows

```
$ simics.bat -stall -c at-workload-start.ckpt
```

Unix

```
$ ./simics -stall -c at-workload-start.ckpt
```

Simics started from graphical user interface:

To select a simulation mode, at the Simics Control window, open **Edit** → **Preferences**, and select the *Startup* icon. Note that any change causes the default mode to change, so the specified execution mode will be the default the next time Simics is started. To force Simics in normal mode again, use the *-fast* command line switch.

Simics does not support switching processor implementations at runtime. To change from one mode to another, use Simics's checkpoint feature to save the simulation state (**File** → **Save Checkpoint As...**) and then restart the session from the saved checkpoint (**File** → **Open Checkpoint**).

3.1.1 Normal mode

Normal mode is the fastest execution mode, and is optimized for emulation-style usage. The following features are not available in Normal mode:

- Instruction and data profiling is not available.
- Attaching timing-models and snoop devices to memory spaces is not supported. This means that the tracing and cache modules do not work correctly, and that stalling is not possible.

3.1.2 Memory Timing with Stall Variant

Stall mode supports stalling and instruction/data profiling.

Note that not all Simics processor models implement a complete support for stalling and profiling.

3.2 Common Options

The following command line options exists when starting Simics. These are the more normal ones. There are also more unusual advanced options listed in the next chapter.

-h
Makes Simics print a brief help screen and exit.

-obscure
Prints out a summary of advanced command line flags and exit.

-no-win
Disable external windows. This will prevent Simics from opening target console windows or any other external windows.

Disabling a target console only makes it invisible. It does not change its simulated functionality. Disabling it is useful when running in batch mode or when the target console isn't needed, since it doesn't require access to a graphical environment, and running with windows disabled is usually somewhat faster.

-gui
Start Simics in command line mode but allow the GUI to open windows.

-no-gui
Start Simics in command line mode. No GUI windows will open except if explicitly requested using CLI commands.

-no-log
Turns off logging of issued CLI commands to the log file. This logging is deprecated and scheduled for future removal.

-log
Turns on logging of issued CLI commands to the `~/.simics/4.6/log` file on Unix and `<APPDATA>/Simics/4.6/log` on Windows (or any other file specified by `-log-file`). Logging is on by default but may have been disabled in the user preferences. This logging is deprecated and scheduled for future removal.

-log-file FILE
Log issued CLI commands to a different file than the default.

-v, -version, --version, -v-short
Print the Simics version number and exit. Other information printed include the compilers used and the compile-time options applied to this specific build. `-v-short` only prints out Simics version.

-license-file FILE
Specify the FlexNet license file to use. This will make Simics ignore the preference setting and the `SIMICS_LICENSE_FILE` environment variable.

-readme
Print the Simics `README` information and exit.

-license
Print the Simics license text and exit. This is the default license and is also included in the `LICENSE` file.

-batch-mode
Run in batch mode. This means that Simics will exit when all commands and scripts given on the command line have executed, or as soon as an error occurs.

When an error has occurred, Simics will immediately exit with a non-zero status. If all commands run to completion, Simics exits with status 0, indicating success.

Simics will not use the command history file in batch mode.

-verbose
Make Simics more verbose. This makes Simics give ample details about the execution. This is the opposite of the `-quiet` flag. The quiet and verbose flags turn off each other. The default is somewhere in the middle with a "reasonable" output level.

Setting the verbose flag can be useful to figure out problems that you may be having with Simics. Some friendly warnings are not printed unless verbose is turned on.

Note that there will be quite a bit of information printed, most of which isn't usually needed.

-fast
"Fast" mode is the default mode of execution. This option can be used to override simulation mode set in preferences.

-stall
Start the stall version of Simics if available. For more information about this version and its limitations, see the *Simulation Modes* section in *Hindsight User's Guide*.

-q or -quiet
Make Simics less verbose. This makes Simics output less information, which is useful for batch execution. The opposite effect can be obtained with the `-verbose` flag. The quiet and verbose flags turn off each other. If you want Simics to be really quiet you can also give the `-no-copyright` flag.

Note that Simics is rather quiet by default, so this flag usually doesn't do all that much.

-c FILE
Load a configuration file. This is equivalent to issuing a **read-configuration** command after Simics has started.

If this flag is the last on the command line, `-c` may be omitted.

-x FILE
Run commands from a script file. The script file is usually named with a `.simics` suffix.

If this flag is the last on the command line, `-x` may be omitted.

-p FILE
Run code from a Python file.

-e COMMAND
Execute a CLI command. Equivalent of typing in the command at the command line prompt.

3.3 Advanced Options

These are more unusual, advanced, command line options to Simics.

-no-settings
Do not read any settings files, such as preferences.

-workspace PATH
Specify the workspace directory to run Simics in.

-wdeprecated
Warn all use of deprecated features.

-wall-deprecated
Warn the use of features that *may* be deprecated in the future. Note that this may not be correct since the actual API is not finalized until the actual release, and Simics itself may still use such features.

-no-wdeprecated
Turn off warnings about deprecated features.

-wmultithread
Warn about non-thread-safe behavior even if Simics is running single-threaded.

-no-wmultithread
Turn off warnings about non-thread-safe behavior (only if Simics is running single-threaded).

-E, -expire TIME
Check out licenses with an expire time, for off-line use. Valid formats are:

- dd-mmm-yyyy[:hh:mm] – where:

  dd
  day number in month

  mmm
  abbreviated name of month

  yyyy
  year including century

  hh
  hour in 24-hour format

  mm
  minutes.

- +<num>h – setting the license to expire <num> hours from now.
- +<num>d – setting the license to expire <num> days from now.

-L
Add a path to the directory list that Simics searches for modules.

-python-verbose
Set the Python verbose flag. This will, for example, make Python print information when Python modules are being loaded. This is the same option as the `-v` flag in `mini-python`.

-werror
Treat many warnings in Simics as errors and exit.

-no-werror
Do not treat warnings in Simics as errors. This is the default behavior but may have been disabled in the user preferences.

-no-stc
Disable the Simics internal caches for memory operations and instruction fetches (STCs) that Simics uses to speed up the simulation.

Disabling the STCs (this flag disables both the D-STC and the I-STC) can be useful for debugging plug-ins such as cache models, since it will ensure that all memory accesses are fully visible.

It is possible to selectively disable the data and instruction STCs using the `-no-dstc` and `-no-istc` flags, respectively. The STCs are typically enabled by default, although in some versions of Simics they may be disabled. See also the `-stc` flag.

-stc
Force the I-STC and D-STC to be enabled (default). See the `-no-stc` flag for more information. The STCs are typically enabled by default, although in some versions of Simics they may be disabled.

-no-istc
Force the I-STC to be disabled. See the `-no-stc` flag for more information.

-no-dstc
Force the D-STC to be disabled. See the `-no-stc` flag for more information.

-istc
Force the I-STC to be enabled. See the `-no-stc` flag for more information.

-dstc
Force the D-STC to be enabled. See the `-no-stc` flag for more information.

-no-copyright
Do not display copyright information when starting Simics. See also the `-verbose` and `-quiet` flags.

-central ADDRESS:[PORT]
This flag is deprecated and will be removed in a future Simics version.

Connect to Simics Central. Simics Central may be running on another host, and this argument to this flag is an IP address and an optional port number where Simics Central is listening. The port number defaults to 1909 if omitted.

If the local host's address is supplied, and the port number is left out, Simics will try to connect to the default Unix file socket (`/tmp/simics-central`).

Also see the `-central :FILE` flag.

-central :FILE
This flag is deprecated and will be removed in a future Simics version.

Connect to Simics Central using a Unix file socket. This variant of the `-central` flag lets you specify a socket file.

-n
Do not run the commands from the `startup-commands` script files.

-echo
Enable command echoing. When echoing is enabled, Simics will echo all commands executed by startup scripts. Note that this only affects any startup scripts loaded after the `-echo` flag on the command line.

-core
Allow Simics to dump core on fatal signals.

-optional-central
This flag is deprecated and will be removed in a future Simics version.

Only use remote Simics Central if set in configuration.

3.4 Environment Variables

These are the environment variables you can set to control how Simics runs. They are all optional, and manually setting them should normally not be needed.

SIMICS_HOST

Overrides the host type detected by Simics. The value must be the name of the directory containing the host-specific files of a Simics installation. Typically a string on the form arch-os, e.g., linux32.

Chapter 4

The Command Line Interface

The Simics Command Line Interface (CLI) is an advanced text based user interface with built-in help system, context sensitive tab-completion, and scripting support (both built-in and using Python). It is provided as part of Simics Hindsight.

If Simics graphical user interface (GUI) is used, the command line interface is accessible via the Simics Control window (`Tools → Command Line Window`). In Simics Eclipse the command line interface is accessible from the Eclipse *Console* view. When running Simics without a GUI, the command line interface directly accessed at the command line/shell where Simics is started.

4.1 Invoking Commands

Commands are invoked by typing them at the command line followed by their arguments. The synopsis part of a command documentation explains how to call a command (you can see many examples in the reference manuals). Here are two examples:

SYNOPSIS
command1 *-x -y -small* [cpu-name] address (size|name)

SYNOPSIS
command2 files ...

Arguments starting with a hyphen are flags and are always optional. Flags can be more than one character long so it is *not* possible to write *-xy* for *-x -y*. The order of the flags is not significant and they can appear anywhere in the argument list.

Arguments enclosed within square brackets are optional; in the example above, it is not necessary to specify *cpu-name*. *address*, on the other hand, is required. The last argument to **command1** is either a size or a name, but not both. Such arguments are called *polyvalues* and can be of different types. Size and name are called sub-arguments.

If an argument is followed by three dots as the file argument in **command2** it indicates that the argument can be repeated one or more times.

The type of the arguments; e.g., whether they are integers or strings, should be evident from their names. For example, *size* should be an integer and *name* a string if not documented otherwise.

Integers are written as a sequence of digits beginning with an optional minus character for negative numbers. Hexadecimal numbers can be written by prefixing them with `0x`, octal numbers with `0o`, and binary numbers with `0b`. Integers may contain "_" characters to make them easier to read. They are ignored during parsing. For example:

```
simics> 170_000
170000
simics> 0xFFFF_C700
4294952704
```

Strings are written as is or within double quotes if they contain spaces or begin with a non-letter. Within double quotes, a backslash (`\`) is an escape character, which can be used to include special characters in the string.

The supported escape sequences are the usual C ones: `\n` for newline, `\t` for tab, `\033` for the octal ASCII character 33 (27 decimal) escape, etc. `\` followed by one, two or three octal digits, or `\x` followed by exactly two hexadecimal digits is the corresponding byte value.

`\u` followed by exactly four hexadecimal digits is the corresponding Unicode character. CLI strings are in the current version of Simics always stored in their UTF-8 encoding, which means that a single `\u` character can be represented as several bytes in the CLI string. As this is expected to change in a future version of Simics, do not rely on this functionality.

```
simics> echo no_quotes_needed
no_quotes_needed
simics> echo "first line\nsecond line"
first line
second line
simics> echo "Two As: \101 \x41, and a micro sign: \u00b5"
Two As: A A, and a micro sign: µ
```

Note: On a **Windows** machine, strings used as paths to files can be written in several different ways. If the path does not contain any blank spaces, it can be written without quotes, using backslashes to separate the directories in the path, like **`C:\temp\file.txt`**. If the path contains spaces, it has to be written within quote characters, and the directory separators have to be written using double backslashes. This is due to the way that the Simics command line uses backslashes to generate special characters like newline and tab. Such a path would look like **`"C:\\Documents and Settings\\user\\My Documents\\file.txt"`**.

Here are some possible invocations of the commands above:

```
simics> command1 -small mpc8641d_simple.soc.cpu 0x7fff_c000 14 -y

simics> command1 0x7fffc000 foo

simics> command1 -x "Pentium 4" 0x7fff_c000 -8
```

```
simics> command2 "/tmp/txt" "../bootdisk" floppy
```

In the first example `cpu-name` is passed as the string `mpc8641d_simple.soc.cpu` and `size` as the integer 14. In the second invocation `cpu-name` has been omitted and `name` is set to the string `foo`. The third example illustrated the use of a string containing a space. In all **command1** examples the address is set to the hexadecimal value `0x7fffc000`. **command2** takes a list of at least 1 string.

A few commonly used commands have aliases. For example, it is possible to write **c** for **continue** and **si** for **step-instruction** for example. Command aliases are documented with their corresponding command in the *Simics Reference Manual*.

4.1.1 How are Arguments Resolved?

Simics tries to match the provided arguments in same order as they appear in the synopsis. If the type of the next argument is identical to what is typed at the command line the argument will match. If there is a mismatch and the argument is optional, the argument will be skipped and the next argument will be matched, and so on. If a mismatching argument is not optional, the interpreter will fail and explain what it expected. For polyvalues, the argument will match if one of its sub-arguments matches.

There are situations however when this method is not sufficient. For example, when two arguments both have the same type and are optional, there is no way to know which argument to match if only one is given. This is resolved by naming the arguments: `arg-name=`*value*. For example **command1** in the example above can be invoked like this:

```
simics> command1 size=32 -y address = 0xf000 -small cpu-name=mpc8641d_simple.soc.cpu[0]
```

Thus there is no ambiguity in what is meant and in fact this is the only way to specify a polyvalue with sub-arguments of the same type. Note also that named arguments can be placed in any order.

4.1.2 Referring to Simics Objects

Many Simics commands accepts configuration object references as arguments. An object reference is simply a string which contains the fully qualified name of the object. Simics provides an hierarchical namespace for objects. The fully qualified name is similar to a file system path, but with the parts separated by `.`-characters. For example, the fully qualified name of `cpu0` in the `system0` namespace is `system0.cpu0`.

Each namespace except the root namespace is also a special kind of configuration object called a component. You can read more about components in section 6.4.

4.1.3 Namespace Commands

Configuration objects (such as devices or CPUs) that define user commands usually place them in a separate namespace. The namespace is the fully qualified name of the object.

Interfaces may also define commands, in which case all objects implementing these interfaces will inherit the commands in their own namespace.

Namespace commands are invoked by typing the fully qualified name of the object, followed by a dot and the command name: *system.component.object.command*, e.g.,

```
simics> system0.board0.cache0.print-status
```

All namespace commands are listed in the *Simics Reference Manual* under the class or interface they belong to.

When using large configurations with an hierarchical structure of components and objects it can be inconvenient to type the fully qualified name all the time when invoking namespace commands. You can than use the **change-namespace** command (alias **cn**) to set a current namespace just like navigating in a file system with the cd command. From the current namespace you can refer to objects with a relative name, for example:

```
simics> cn system0.board0
simics:system0.board0> cache0.print-status
```

Note that the Simics prompt changes to reflect the new position in the hierarchy. You can only change the current namespace to other components. It would have been illegal to do:

```
simics> cn system0.board0.cache0
system0.board0.cache0 is not a component
```

Cache0 is not a component, it is an object located in the cache0 slot (see 6.4 and 6.5 for more information on how objects are named and referenced).

To go "up" one level in the hierarchy you can type **cn ..**:

```
simics:system0.board0> cn ..
simics:system0>
```

You can still refer to other components relative to the root by writing a dot before the fully qualified name, e.g.:

```
simics:system0> .system1.cpu0.ptime
processor     steps  cycles  time [s]
system1.cpu0  14545   14545     0.000
```

The command **current-namespace** is provided to get the current namespace and can be used in scrips to save a location in a CLI variable:

```
simics:system0> current-namespace
.system0
simics:system0> $location = (current-namespace)
simics:system0> cn ..
```

```
simics> cn $location
simics:system0>
```

See section 5 for more information on Simics scripting and CLI variables.

4.1.4 Expressions

The CLI allows expressions to be evaluated, for example:

```
print -x 2*(0x3e + %g7) + %pc
```

The precedence order of the operators is as follows (highest first):

`$`	variable access
`%`	register access
`[]`	list indexing
`->`	attribute access
`pow`	power of
`~`	bitwise not
`*, /, %`	multiplication, division, modulo
`+, -`	addition, subtraction
`<<, >>`	left, right shift
`&`	bitwise and
`^`	bitwise xor
`|`	bitwise or
`<, <=, ==, !=, >=, >`	comparison
`not`	boolean not
`and`	boolean and
`or`	boolean or

Parentheses can be used to override the priorities. Commands which return values can also be used in expressions if they are enclosed within parentheses:
`print -x (mpc8641d_simple.soc.cpu[0].read-reg g7)`

Values can be saved in variables for later use. You set a variable by simply giving an assignment command such as **$var = 15**. You can also store a command in a variable such as **$my_read_reg = mpc8641d_simple.soc.cpu[0].read-reg** which is different from storing the return value from a command **$value_g7 = (mpc8641d_simple.soc.cpu[0].read-reg g7)**.

4.1.5 Interrupting Commands

GUI

Use the **stop** command at the `running>` prompt or the *stop* button in *Simics Control window*.

CLI without GUI

Any command which causes the simulation to advance can be interrupted by typing **control-C**. The simulator will gracefully stop and prompt for a new command. If Simics

hangs for some reason, possibly due to some internal error, you can usually force a return to the command line by pressing **control-C** two or more times in a row.

Note: Pressing **control-C** several times may damage some internal state in the simulator so should be used as a last resort.

4.2 Tab Completion

The command line interface has a tab-completion facility. It works not only on commands but on their arguments as well. The philosophy is that the user should be able to press the tab key when uncertain about what to type, and Simics should fill in the text or list alternatives.

Note: Tab completion on a **Windows** host does not work when running in a Cygwin terminal. Only from the graphical user interface or from a Windows command line console.

For example **com**<tab> will expand to the command beginning with **com** or list all commands with that prefix if there are several. Similarly, **disassemble** <tab> will display all arguments available for the command. In this case Simics will write:

```
address =   count =     cpu-name =
```

to indicate that these alternatives for arguments exists. Typing **disassemble cp**<tab> will expand to **disassemble cpu-name =** and a further tab will fill in the name of the CPU that is defined (or list all of them).

4.3 Help System

The most useful Simics commands are grouped into categories. To list these categories, just type **help** at the command prompt. The list should look like this:

```
simics> help
[...]
To get you started, here is a list of command categories:
  Breakpoints                  Files and Directories        Python
  CD-ROM                       GUI                          Real Network
  Changing Simulated State     Haps                         Registers
  Command Line Interface       Help                         Reverse Execution
  Components                   Inspecting Simulated State   Simics Search Path
  Configuration                Logging                      Speed
  Debugging                    Memory                       Symbolic Debugging
  Disk                         Modules                      Test
  Distributed Simulation       Networking                   Tracing
  Ethernet                     Output
  Execution                    Profiling
```

```
[...]
```

Note that since Simics's configuration can change between sessions and even dynamically through loading modules, the commands and command categories may look different.

Type **help** ***category*** for a list of commands, e.g., **help "Changing Simulated State"** will list all commands belonging to that category:

```
simics> help "Changing Simulated State"
Commands available in the Changing Simulated State category:

  <image>.set                  set bytes in image to specified value
  <memory-space>.load-binary   load binary (executable) file into memory
  <memory-space>.load-file     load file into memory
  <memory-space>.set           set physical address to specified value
  load-binary                  load binary (executable) file into memory
  load-file                    load file into memory
  pdisable                     switch processor off
  penable                      switch processor on
  set                          set physical address to specified value
  set-pc                       set the current processor's program counter
  write-reg                    write to register
```

Type **help** ***command*** to print the documentation for a specific command.

The **help** command can do much more than printing command documentation: it gives you access to nearly all Simics documentation on commands, classes, modules, interfaces, API types and functions, haps and more according to the configuration loaded in the simulator. All documentation is also available in the reference manuals.

Here are some more examples of usage of the **help** command:

```
simics> help ptime
[... ptime command documentation ...]

simics> help mpc8641d_simple.soc.cpu[0].disassemble
[... <processor_info>.disassemble command documentation ...]

simics> help <processor_info>.disassemble
[... <processor_info>.disassemble command documentation ...]

simics> help mpc8641d_simple.soc.cpu[0].cpu
[... <ppce600> class documentation ...]

simics> help ppce600
[... <ppce600> class documentation ...]

simics> help processor
```

```
[... <processor> interface documentation ...]

simics> help processor_info
[... <processor_info> interface documentation ...]

simics> help mpc8641d_simple.soc.cpu[0].gprs
[... <ppc440gp>.gprs attribute documentation ...]

simics> help ppce600.gprs
[... <ppce600>.gprs attribute documentation ...]

simics> help Core_Exception
[... Core_Exception hap documentation ...]

simics> help SIM_get_mem_op_type
[... SIM_get_mem_op_type() function declaration ...]

simics> help ppce600-turbo
[... ppce600-turbo module documentation ...]
```

When a name matches several help topics (for example, a command and an attribute, or a module and a class), **help** will print out the first topic coming in this order: command categories, commands, classes, interfaces, haps, modules, attributes, API functions and symbols. It will also inform you at the end of the documentation output that other topics were matching your search:

```
simics> load-module NS16550_c
simics> help NS16550_c
[... NS16550_c class documentation ...]

Note that your request also matched other topics:
  module:NS16550_c
```

If you type **help module:NS16550_c**, the module documentation will be printed instead:

```
simics> help module:NS16550_c
[... NS16550_c module documentation ...]
```

You can use specifiers like `module:` or `class:` at any time. It will also allow the **help** command to provide you better tab-completion, since only items in the selected category of documentation will be proposed. The following specifiers are available: `object:`, `class:`, `command:`, `attribute:`, `interface:`, `module:`, `api:`, `hap:` and `category:`.

Note: By default, **help** does not propose tab-completion for modules and API symbols, because they tend not to be the most searched for and would clutter the tab-completion propositions unnecessarily. You can get tab-completion for those by specifying `module:` or `api:` in front of what you are looking for.

The **help-search** command can search for keywords in the documentation provided by **help**. Type **help-search** ***keyword*** to get a list of all documentation topics matching this keyword. Its alias is **apropos**, as the Unix model is named.

```
simics> help-search step
The text 'step' appears in the documentation
for the following items:

Command         <processor>.cycle-break-absolute
Command         <processor>.step-break
Command         <processor>.step-break-absolute
Command         cycle-break-absolute
Command         log-setup
Command         print-event-queue
[...]
Attribute       <ultrasparc-iii>.steps
Hap             Core_Step_Count

simics> apropos step
[... yields the same output ...]
```

4.4 Simics's Search Path

Many Simics commands will look up files based on the current directory. When Simics is launched from the command line the current directory is the current directory of the shell Simics was launched from, and when Simics is launched by double clicking its icon or from Simics Eclipse it is the workspace directory. This may be impractical when writing scripts or building new configurations, so Simics provides two features to ease directory handling:

- Simics recognizes some special path markers that are translated before being used:

  `%simics%`
  This path marker causes Simics to locate the file (following the marker) in all installed packages. Simics searches for the file starting in the newest (highest build-id number) package and continues to find the file until it has reached the oldest (lowest build-id number) package.
  For example, `%simics%/scripts/foo.simics` could be translated to:
  Windows:
  `C:\Program Files\Simics\Simics 4.4\Simics 4.4.1\scripts\foo.simics`

Unix:

`/opt/simics/simics-4.4/simics-4.4.1/scripts/foo.simics`

Note that if you change the version of Simics you are using, `%simics%` will change as well, so you should use it to refer only to files that you know are present in all Simics versions. Notice also that `%simics%` has no meaning if no file can be found. You can use the command **lookup-file** to find out how the path will be translated.

`%script%`

Translated to the directory where the currently running script is located. A possible usage is to let a script call another one in the same directory, independently of what the current directory is.

For example, if the directory `baz` contains the scripts `foo.simics` and `bar.simics`, even if the user uses Simics with another current directory, it will be possible for `foo.simics` to call `bar.simics` by issuing the command:

```
run-command-file %script%/bar.simics
```

`%simics%` and `%script%` are always translated to absolute paths, so they never interact with the next feature, called *Simics's search path*. One consequence is that they must always be used in double quotes `"%simics%/targets/mpc8641-simple/images"` to ensure that escaped characters such as spaces are used correctly.

- Simics has a list of paths called *Simics's search path* where files will be looked up when using some specific commands (among others, **load-binary**, **load-file**, **run-command-file**, and **run-python-file**) and a number of classes (such as the **image** class and the tftp server implementation in the **service-node**). The file is first looked up in the current directory, then in all entries of Simics's search path, in order.

  Windows:

  Let us assume for example that Simics's search path contains
  `C:\Documents and Settings\joe\scripts\`
  and that the current directory is
  `C:\Documents and Settings\joe\workspace\`
  If the command:

  ```
  simics> run-command-file test/start-test.simics
  ```

  is issued, Simics will look for the following files to run:

  1. `C:\Documents and Settings\joe\workspace\test\start-test.simics`
  2. `C:\Documents and Settings\joe\scripts\test\start-test.simics`

  Unix:

  Let us assume for example that Simics's search path contains
  `/home/joe/scripts/` and that the current directory is `/home/joe/workspace`.
  If the command:

  ```
  simics> run-command-file test/start-test.simics
  ```

  is issued, Simics will look for the following files to run:

  1. `/home/joe/workspace/test/start-test.simics`

2. `/home/joe/scripts/test/start-test.simics`

Simics's search path can be manipulated using the **add-directory**, **clear-directories** and **list-directories** commands. Simics's search path is also used when looking for images files belonging to checkpoints or new configuration. This is described in section 6.2.3.

Remember that setting a CLI variable to a path with %simics% or %script% does not in itself evaluate the path marker. This means that the path marker may evaluate to another directory than is anticipated. The following two lines evaluate quite differently:

```
$just_a_string = "%script%/images/my_image"
$absolute_path = (resolve-file "%script%/images/my_image")
```

The first CLI variable reads the given text and may evaluate to wherever. The second variable is evaluated locally and reads the absolute path for the `my_image` file that is located nearby the script.

Note: Although the Simics search path is saved in the `sim` object in checkpoints, allowing image files that were found through it to be opened again by the checkpoint, it is not available until the object creation phase. Module initialization code should not rely on the Simics path since that code is run before the `sim` object from the checkpoint has been created.

4.5 Using the Pipe Command

The **pipe** command lets you send the output of a Simics command to a shell command through a pipe:

```
simics> pipe "help" "grep Tracing"
```

This will run **help** (which lists all Simics commands categories) and send its output to the standard input of the **grep trace** process. **grep** will discard all lines not containing "Tracing" and forward the rest to its standard output, which will be printed on the Simics terminal.

The **pipe** command can be used to send all the output of a command to a file:

```
simics> pipe "stepi 1000" "cat > trace.txt"
```

Or you can use it to view information using the shell command **more**:

```
simics> pipe "pregs -all" more
```

Note that you have to enclose both the Simics command (the first argument) and the shell command (the second argument) in double quotes if they contain whitespace or other non-letter characters.

4.6 Running Shell Commands

The **!** Simics command can be used to run command line commands. It will take everything after the **!** sign and run it in a command interpreter (the current shell, on Unix, and `cmd.exe`, on Windows). For example:

Windows

```
simics> !dir c:\
```

This will run **dir c:** as if typed at the **cmd.exe** prompt and show the result in the Simics console.

Unix

```
simics> !uname -a
```

This will run **uname -a** using the current shell and show the result in the Simics console.

4.7 Command-line Keyboard Shortcuts

The Simics command line supports two modes with different keyboard shortcuts: *Windows* and *GNU Readline* style. Most shortcuts are the same in both modes but there are some minor differences between the two as listed in the following table. The mode to use can be set in the preferences. Either in the GUI or at the command line:

```
simics> prefs->readline_shortcuts = TRUE
simics> save-preferences
```

The command line in Simics can be accesses in several different ways. The shortcuts are supposed to be the same everywhere, but some terminals and telnet clients may not forward certain key combinations to Simics. A typical example where keyboard shortcuts do not work properly is the Cygwin terminal on Windows hosts. To run Simics in command-line mode on Windows, a standard Windows command line console is recommended.

4.7.1 List of Shortcuts

The following is a list of all keyboard shortcuts supported in Simics, where some are marked as Windows or GNU Readline only.

Move Shortcuts

Action	Shortcuts
Move character left	`Ctrl-B, Left`
Move character right	`Ctrl-F, Right`
Move word left	`Alt-B, Ctrl-Left`
Move word right	`Alt-F, Ctrl-Right`
Move to start of line	`Ctrl-A` (GNU Readline), `Home`
Move to end of line	`Ctrl-E, End`

Edit Shortcuts

Action	Shortcuts
Enter line	`Ctrl-J, Ctrl-M, Enter`
Copy	`Ctrl-C, Ctrl-Insert`
Paste	`Ctrl-Y` (GNU Readline), `Ctrl-V, Shift-Insert`
Cut	`Ctrl-X, Shift-Delete`
Cut to end of line	`Ctrl-K`
Cut to start of line	`Ctrl-U`
Cut previous word	`Ctrl-W`
Select character left	`Shift-Left`
Select character right	`Shift-Right`
Select word left	`Ctrl-Shift-Left`
Select word right	`Ctrl-Shift-Right`
Select to start of line	`Shift-Home`
Select to end of line	`Shift-End`
Select line	`Ctrl-A` (Windows)
Delete character left	`Ctrl-H, Backspace`
Delete character right	`Ctrl-D, Delete`
Delete word left	`Ctrl-Backspace, Alt-Backspace, Alt-Delete`
Delete word right	`Alt-D, Ctrl-Delete`
Delete to start of line	`Ctrl-Home`
Delete to end of line	`Ctrl-End`

History Shortcuts

Action	Shortcuts
Next in history	`Ctrl-N, Down`
Previous in history	`Ctrl-P, Up`
First in history	`Alt-<, Page Up`
Last in history	`Alt->, Page Down`
Reverse search	`Ctrl-R`
Scroll page up	`Shift-Page Up`
Scroll page down	`Shift-Page Down`

Completion Shortcuts

Action	Shortcuts
Auto complete	`Ctrl-I, Tab`
Show completions	`Alt-?`

Transpose Shortcuts

Action	Shortcuts
Uppercase word	`Alt-U`
Lowercase word	`Alt-L`
Capitalize word	`Alt-C`
Transpose characters	`Ctrl-T`
Transpose words	`Alt-T, Ctrl-Shift-T`

Undo Shortcuts

Action	Shortcuts
Cancel multi-line editing	`Ctrl-G`
Undo	`Ctrl-_, Ctrl-Z` (Windows)
Revert line	`Alt-R`
Clear screen	`Ctrl-L`

Chapter 5

Simics Scripting Environment

The Command Line Interface in Simics has simple scripting capabilities that can be used when writing parameterized configurations and scripts with conditional commands. For more advanced scripting, the Python language can be used.

This chapter describes how to write simple scripts in the Simics command line interface (CLI), using control flow commands, and variables. It also explains how the configuration system can be accessed from scripts, and how Python can be used for more advanced scripting.

All commands can be executed either by typing them at the prompt in the Simics console, or by writing them to a file, e.g. `example.simics`, and executing the command `run-command-file example.simics`, or for Python examples: `run-python-file example.py`.

5.1 Script Support in CLI

5.1.1 Variables

The Simics command line has support for string, integer, floating point, list and boolean variables. Variables are always prefixed with the `$` character. Using undefined variables is deprecated.

```
simics> $foo = "some text"
simics> $foo
"some text"
simics> echo $not_used_before
+++ Use of uninitialized CLI variable (not_used_before) is deprecated.
Initialize variable before use.
0
```

The **defined** expression can be used to test if a variable has been defined. Note that this command takes the name of the variable only, i.e. without the `$`.

```
simics> $foo = 4711
```

```
simics> if defined foo { echo "foo is defined"}
foo is defined
```

List variables can be indexed, something that is useful in loops for example.

```
simics> $foo[0] = []
simics> $foo[0] = 10
simics> $foo[1] = 20
simics> echo $foo[0] + $foo[1]
30
simics> $foo
[10, 20]
simics> $foo += ["abc"]
[10, 20, "abc"]
simics> list-length $foo
3
```

CLI also has support for local variables, described later in this chapter.

5.1.2 Command Return Values

The return value of a command is printed on the console, unless it is used as argument to some other command. Parenthesis `()` are used to group a command with arguments together, allowing the return value to be used as argument. The return value can also be used as namespace in another command. Variables can be used in the same way.

```
simics> $address = 0xff800000
simics> set $address 20
simics> echo "The Value at address " + $address + " is " + (get $address)
The Value at address 0 is 20

simics> $id = 0
simics> ("mpc8641d_simple.soc.cpu[" + $id + "]").print-time
processor                   steps  cycles  time [s]
mpc8641d_simple.soc.cpu[0]      0       0     0.000
```

Although in this particular case it is simpler to write:

```
simics> mpc8641d_simple.soc.cpu[$id].print-time
processor                   steps  cycles  time [s]
mpc8641d_simple.soc.cpu[0]      0       0     0.000
simics> $cpu = mpc8641d_simple.soc.cpu[0]
simics> $cpu.print-time
processor                   steps  cycles  time [s]
mpc8641d_simple.soc.cpu[0]      0       0     0.000
```

Parenthesis can also be used to enter a multi-line command, making it easier to read scripts with nested command invocations. In the text console, the prompt will change to `.......` for code spanning more than one line.

```
simics> (echo 10
.......       + (20 - 5)
.......       + (max 4 7))
32
```

5.1.3 Control Flow Commands

The script support in CLI has support for common flow control commands such as **if**, **else**, **while** as well as **foreach**.

```
simics> $value = 10
simics> if $value > 5 { echo "Larger than five!" }
Larger than five!
```

The **if** statement has a return value:

```
simics> $num_cpus = 2
simics> (if $num_cpus > 1 { "multi" } else { "single" }) + "-pro"
multi-pro
```

Note: Multi-line **if-else** statements must have `} else {` on the same line.

It is also possible to have else followed by another if statement.

```
simics> $b = 0
simics> if $b == 1 {
.......     echo 10
....... } else if $b == 0 {
.......     echo 20
....... } else {
.......     echo 30
....... }
20
```

Loops can be written with the **while** command.

```
simics> $loop = 3
simics> while $loop {
.......     echo $loop
.......     $loop -= 1
```

```
....... }
3
2
1
```

They can also be written by using the **foreach** list iteration command. The **range** commands in the example returns a list of integers from 0 up to, but not including, 3.

```
simics> foreach $loop in (range 3) {
.......     echo $loop
....... }
0
1
2
```

Here is another example that shows how **foreach** can be used. The **get-object-list** commands return a list of all objects that implement the processor interface in Simics:

```
simics> foreach $cpu in (get-object-list -all processor) {
.......     echo "Cycles on " + ($cpu->name) + ": " + ($cpu.print-time -c)
....... }
Cycles on mpc8641d_simple.soc.cpu[0]: 0
Cycles on mpc8641d_simple.soc.cpu[1]: 0
```

Lists can also be written directly, for example:

```
simics> foreach $loop in [1, 2, 3] {
.......     echo $loop
....... }
1
2
3
```

Within command blocks, it can be useful to have variables that are local to the scope and thus do not collide with the names of global variables. By adding **local** before the first assignment of a variable, the variable is made local.

```
simics> $global = 10
simics> if TRUE {
.......     local $global = 20
.......     echo $global
....... }
20
simics> echo $global
```

```
10
```

5.1.4 Integer Conversion

In some cases it is useful to interpret an integer as a signed value of a specific bit size, for example when reading four bytes from memory that should be interpreted as a signed 32 bit integer. The **signed8**, **signed16**, ..., **signed64** commands can be used in to perform the conversion.

```
simics> mpc8641d_simple.soc.phys_mem.set 0xf8400000 0xffffffff 4
simics> mpc8641d_simple.soc.phys_mem.get 0xf8400000 4
4294967295
simics> signed32 (mpc8641d_simple.soc.phys_mem.get 0xf8400000 4)
-1
```

5.1.5 Accessing Configuration Attributes

Simics configuration attributes that are of string, integer, floating point, boolean, nil, and list types can be accessed directly from CLI using the **->** operator.

```
simics> echo "Current workspace: " + (sim->workspace)
Current workspace: C:\Documents and Settings\joe\Desktop\workspace
```

An object referenced with this operator returns the object's name as a string.

A nil attribute value is represented by zero when read by CLI. You cannot assign nil to an attribute from CLI.

To access attributes that use other data types than the ones listed above, you need to use Python:

```
simics> @conf.myobject.dictionary_attribute = { 1 : "abc" }
```

See chapter 5.2.3 for more information about accessing attributes from Python.

5.1.6 Script Branches

Introduction to Script Branches

Script branches allow the user to write sequences of CLI commands that can wait for Simics haps (see chapter 5.2.6) at anytime without breaking the sequential flow of commands. This is typically used to avoid breaking a script into many small sections, each installed as a hap callback written in Python.

A simple example from a Simics script:

```
script-branch {
```

```
    echo "This is a script branch test - going to sleep."
    mpc8641d_simple.soc.cpu[0].wait-for-step 10
    echo "Processor registers after 10 steps:"
    mpc8641d_simple.soc.cpu[0].pregs
}
```

The example above will execute the first **echo** command at once, and then go to sleep waiting until the first 10 instructions (steps) have run. When the step counter for the processor has reached 10, the branch will wake up and run the next two commands, **echo** and **pregs**.

Some commands can not be run while Simics is executing. One example is the **write-configuration** command. To issue such commands from a script branch, it is possible to stop the execution, issue the command and then resume the simulation. The following is an example that writes a checkpoint when the simulation reaches a login prompt, and then continues running. It assumes that a text-console called `text_console_cmp0.con` is used.

```
script-branch {
    mpc8641d_simple.console0.con.wait-for-string login
    stop
    write-configuration login.conf
    run
}
```

How Script Branches Work

When a script branch is started (using **script-branch**), it begins executing immediately, and runs until a **wait-for-**, command is issued. Execution is then resumed in the main script; i.e., there is never any concurrent activity. When a hap, or some other activity, occurs that a script branch is waiting for, the branch continues executing once the currently simulated instruction is ready.

Note: Since only one branch can be active at once, any callback to Python from Simics will execute in the currently active branch, i.e., if a branch installs a callback, it is most likely that it will be called when the main branch is active.

Script Branch Commands

The following is a list of the commands related to script branches.

script-branch
Create a new script branch and start it.

list-script-branches
List all existing, but suspended, branches.

interrupt-script-branch
Interrupt a script-branch, causing it to exit.

create-script-barrier *num_branches*
Create a script barrier used to synchronize the execution of several script branches. The argument is the number of script branches that must enter the barrier before all of them are released.

wait-for-script-barrier *barrier*
Suspend branch until enough branches have reached the script branch barrier.

create-script-pipe
Create a script pipe, used to communicate data between script branches and also to synchronize them.

add-data-to-script-pipe *pipe data*
Add data (integer, string, floating point value or a list of any of those types) to the specified script pipe.

script-pipe-has-data *pipe*
Check if there is data to read from a script pipe.

wait-for-script-pipe *pipe*
Suspend branch until there is data to read on a script pipe. If there already is data in the pipe, return immediately. The return value is the first data item added to the pipe.

wait-for-breakpoint *breakpoint-id*
Suspend branch until a specified breakpoint is triggered.

<**processor**>**.wait-for-cycle** *cycle* **[**-*relative***]**
Suspend branch until the specified *cycle* on the processor has been reached. If *-relative* is specified, the branch will be suspended until the processor has executed the specified number of cycles instead.

<**processor**>**.wait-for-step** *step* **[**-*relative***]**
Suspend branch until the specified *step* on the processor has been reached. If *-relative* is specified, the branch will be suspended until the processor has executed the specified number of steps instead.

<**processor**>**.wait-for-time** *seconds* **[**-*relative***]**
Suspend branch until the specified time in *seconds* on the processor has been reached. If *-relative* is specified, the branch will be suspended until the processor has executed the specified number of seconds instead.

<**int_register**>**.wait-for-register-read** *reg*
Suspend branch until the register *reg* in the object implementing the `int_register` interface is read. Only registers that catchable can be waited on.

<**int_register**>**.wait-for-register-write** *reg*
Suspend branch until the register *reg* in the object implementing the `int_register` interface is written. Only registers that catchable can be waited on.

<**text-console**>**.wait-for-string** *string*
Suspend branch until *string* is printed on the text console.

Variables in Script Branches

Variable references in CLI are evaluated when accessed. This is important to remember when writing script branches, since some commands are executed when the branch has been suspended, and variables may have been changed. To make sure that CLI variables in script branches are untouched by other scripts, they should be made local.

The following example

```
script-branch {
    $foo = 20
    mpc8641d_simple.soc.cpu[0].wait-for-step 10
    echo "foo is " + $foo
}
$foo = 15
run
```

will produce the output `foo is 15` while the following script will print `foo is 20`.

```
script-branch {
    local $foo = 20
    mpc8641d_simple.soc.cpu[0].wait-for-step 10
    echo "foo is " + $foo
}
$foo = 15
run
```

Canceling Script Branches

It is possible to cancel a suspended script branch by interrupting it using the **interrupt-script-branch** command. Each branch has an ID associated that can be found using **list-script-branches**, and that is returned by the **script-branch** command.

```
$id = (script-branch {
    wait-for-breakpoint $bp
})

...

simics> interrupt-script-branch $id
Command 'wait-for-breakpoint' interrupted.
Script branch 1 interrupted.
```

Script Branch Limitations

There are some limitations to script branches. The first two in the list are enforced by Simics:

- Script branches may not start new branches.
- The main branch may not issue the **wait-for-** commands.
- Breaking the simulation with multiple control-C, which forces Simics back to prompt, may confuse the Python interpreter about what thread is active. (Interrupting Simics this way typically destroy the simulation state anyway.)

5.1.7 Error Handling in Scripts

When a Simics command encounters some kind of error, an error message is typically printed and the script execution is interrupted. In some cases the script itself want to handle the error to be able to try some alternative or to present the error message with more context. The **try-except** statement can be used for this purpose.

```
try {
    load-module my-own-components
} except {
    echo "Simics failed to import my-own components. Perhaps you forgot to "
    echo "install the latest modules from the development team? See the "
    echo "project web-site for more info.\n"

    interrupt-script "Cannot continue"
}
```

Without the **try-except** statement, the example above would print an error message like `Error loading module` and the script execution would be interrupted with an error.

The error message from the failing command can be obtained inside the **except** block by calling the **get-error-message** CLI command. The **get-error-line** command return the line of the error in the script file and **get-error-file** the file name. The **get-error-command** returns the command name if the error occurred within a command.

5.2 Scripting Using Python

Simics provides support for the script language Python (`http://www.python.org`). By using Python the user can extend Simics, and control it in greater detail. Python can interface with Simics using functions in the *Simics API*.

5.2.1 Python in Simics

Python is normally hidden from the user by the command line interface (CLI). But since CLI is implemented in Python, it also provides simple access to the Python environment, making it easy to write your own functions and scripts.

Note: All commands in Simics are implemented as Python functions. Commands for modules are often included in the corresponding source-code package. In that case you can find the source for the commands for device modules in *[src-package]*`/src/devices/`*[module-name]*`/commands.py` or *[src-package]*`/src/devices/`*[module-name]*`/gcommands.py` and for extension modules in *[src-package]*`/src/extensions/`*[module-name]*`/commands.py` or *[src-package]*`/src/extensions/`*[module-name]*`/gcommands.py`, where *[src-package]* is the location of the source package and *[module-name]* is the name of the module.

To execute some Python code from the command line, prefix the line with the @ character. Example:

```
simics> @print "This is a Python line"
This is a Python line
simics>
```

For code spanning more than one line, the prompt will change to `.......` and more code can be inserted until an empty line is entered. The full code block will then be executed. (Note that whitespace indentation is significant in Python.) Example:

```
simics> @if SIM_number_processors() > 1:
.......     print "Wow, an MP system!"
....... else:
.......     print "Only single pro :-("
.......
Wow, an MP system!
simics>
```

Entering more than one line is useful for defining Python functions. It is also possible to execute Python code from a file, which is done with the **run-python-file** command.

If the Python code is an expression that should return a value to the CLI, the `python` command can be used, or the expression can be back-quoted. The following example selects a file with Python commands to execute depending on the number of processors in the system:

```
simics> run-python-file `"abc-%d.py" % SIM_number_processors()`
```

If the system has 2 processors, the file `abc-2.py` will be executed.

5.2.2 Accessing CLI Variables from Python

CLI variables can be accessed from Python via the `simenv` name space, for example:

```
simics> $cpu = "processor"
simics> @simenv.cpu = simenv.cpu.capitalize()
simics> $cpu
Processor
```

5.2.3 Accessing the Configuration from Python

Configuration Objects

All configuration objects are visible as objects in Python. The global Python module `conf` holds the top level namespace, which contains all top level objects, including namespace objects. The namespace objects contain all their subobjects as attributes. All objects also exposes their Simics attributes as Python attributes, which means they can be both read and written as Python attributes. Example: (print the *pci_devices* attribute in a **pci-bus** object that is called **pcibus25B** in the machine where the example was taken from)

```
simics> @print conf.mpc8641d_simple.soc.pcie_bus[0].pci_devices
[[0, 0, <the pex8111 'mpc8641d_simple.pci_bridge'>, 1]]
```

If a namespace object contains an attribute and a subobject with the same name, the attribute takes precedence and hides the subobject.

To try this example in an arbitrary configuration, run **list-objects pci-bus** to find possible **pci-bus** objects to use instead of **pcibus**.

Any '−' (dash) character in the object name, or in an attribute name, is replaced by '_' (underscore). This substitution is performed because Python always treats the dash character as the minus operator. To avoid confusion the recommendation is to always use underscore.

Indexed attributes can be accessed using `[]` indexing in Python. It is also possible to index other list attributes this way, but it might be inefficient since the full list is converted to a Python list before the element is extracted. Here are some examples of indexed attributes access (a **i2c** object, and a **memory-space** object):

```
simics> @print conf.mpc8641d_simple.soc.i2c_bus[1].i2c_devices[0]
['mpc8641d_simple.memory0.sdram_spd', 81]

simics> @print conf.mpc8641d_simple.soc.phys_mem.memory[0x1000:0x100f]
(126, 144, 67, 166, 126, 177, 67, 166, 126, 128, 0, 38, 58, 161, 255)

simics> @conf.mpc8641d_simple.soc.phys_mem.memory[0x10000:0x10003] = (100, 101, 102)
```

Warning: Python only supports 32 bit integers in keys when doing sliced indexing (no long integers). However, the Simics API treats [m:n] synonymous to [[m, n-1]], so instead of `conf.mpc8641d_simple.soc.phys_mem.memory[0x1fff80082f0:0x1fff80082f8]` (which will not work), write `conf.mpc8641d_simple.soc.phys_mem.memory[[0x1fff80082f0,0x1fff80082f7]]`

If the attribute contains a list, dictionary or data, then an access returns a reference instead of the actual value. This is similar to how container objects such as lists and dictionaries work in Python and allows constructs such as:

```
@conf.mpc8641d_simple.soc.phys_mem.map[0][0] = 0x1000
```

Before Simics 3.2, that example would simply return a copy of the map attribute that was modified and then thrown away. Starting with Simics 3.2 the attribute is modified at position [0][0].

To get the copy of the attribute value, the following can be used:

```
memory_map = conf.mpc8641d_simple.soc.phys_mem.map.copy()
```

Note that there is a difference in how references to Simics attributes work compared to ordinary Python objects: if the attribute access returns a list, dictionary or tuple, then a reference to the full attribute is used and not only to the referenced container objects. The reason is that internally in Simics, the attribute is treated as a single value.

Consider a list of lists, such as `a = [[1, 2, 3], [4, 5, 6]]`. If this was a Python list, then the following applies:

```
b = a[0]       # b is a reference to the [1, 2, 3] list.
a[0][1] = 9    # b will now change to [1, 9, 3].
a[0] = [7, 8]  # b still references the [1, 9, 3] list, only a will change.
```

If `a` instead is a Simics attribute:

```
b = a[0]       # b is a reference to the first list in a, i.e. [1, 2, 3].
a[0][1] = 9    # b will now change to [1, 9, 3].
a[0] = [7, 8]  # b is still a reference to the fist list in a, i.e [7, 8].
```

As we see, only the last line of the examples differ. The most common situation where this difference is visible is when doing list duplication. In Python a list can be duplicated in whole or part by using slicing to produce a shallow copy. In Simics, that would simply produce a reference to the same list if any of the items in the list is a container object. In this case the `.copy()` method has to be used.

Interfaces

From Python, the *iface* attribute of a configuration object can be used to access the interfaces it exports. Use *obj*`.iface.`*name*`.`*method* to accesses the *method* function in the *name* interface of the *obj* object. Example:

```
simics> @conf.mpc8641d_simple.soc.cpu[0].iface
<interfaces of mpc8641d_simple.soc.cpu[0]>
simics> @conf.mpc8641d_simple.soc.cpu[0].iface.processor_info
<processor_info_interface_t interface of mpc8641d_simple.soc.cpu[0]>
simics> @conf.mpc8641d_simple.soc.cpu[0].iface.processor_info.get_program_counter
<built-in method logical_address_t (*)([conf_object_t *]) of interface
method object at 0x39142f0>
```

```
simics> @"%x" % conf.mpc8641d_simple.soc.cpu[0].iface.processor_info
.get_program_counter()
'fff00100'
```

Note: When called from Python, the first `conf_object_t *` argument has been deprecated (as of Simics 4.0) and should not be used for interface methods.

The last command corresponds to the following C code (with no error-checking):

```
conf_object_t *obj = SIM_get_object("mpc8641d_simple.soc.cpu[0]");
processor_info_interface_t *iface =
        SIM_get_interface(obj, PROCESSOR_INFO_INTERFACE);
logical_address_t pc = iface->get_program_counter(obj);
printf("%llx", pc);
```

Port Interfaces

Port interfaces are accessed from Python in a similar way to interfaces.

Use *obj*`.ports.`*portname*`.`*interfacename*`.`*method* to access the *interfacename* interface in port *portname* of the object *obj*. Example:

```
simics> @conf.mpc8641d_simple.soc.cpu[0].ports
<ports of mpc8641d_simple.soc.cpu[0]>
simics> @conf.mpc8641d_simple.soc.cpu[0].ports.HRESET
<interfaces of port HRESET of mpc8641d_simple.soc.cpu[0]>
simics> @conf.mpc8641d_simple.soc.cpu[0].ports.HRESET.signal
<signal_interface_t interface of mpc8641d_simple.soc.cpu[0]>
simics> @conf.mpc8641d_simple.soc.cpu[0].ports.HRESET.
signal.signal_raise
<built-in method void (*)([conf_object_t * NOTNULL]) of
interface method object at 0x2b75090>
simics> @conf.mpc8641d_simple.soc.cpu[0].ports.HRESET.
signal.signal_raise()
```

The last command corresponds to the following C code:

```
conf_object_t *obj = SIM_get_object("mpc8641d_simple.soc.cpu[0]");
signal_interface_t *iface =
        SIM_get_port_interface(obj, SIGNAL_INTERFACE, "HRESET");
iface->signal_raise(obj);
```

Configuration Attributes

As there is an automatic translation between Python data types and Simics configuration attribute data types, one can usually assign a Python value (of the appropriate type) to a Simics configuration attribute. For example:

```
simics> @conf.sim.time_quantum = 1e-3
```

However, there are circumstances when this does not work. For example, the hexadecimal floating-point notation is not supported by the Python version Simics uses, and there is no way to represent an invalid attribute value in Python.

To handle these cases, the Python module `configuration` contains a few classes that create special attribute values that can be used in attribute assignments. The *Simics Reference Manual* describes the classes in detail. Here is an example of how they can be used:

```
simics> @import configuration
simics> @conf.foo.float_attr = configuration.FLOAT("0xc.90fep-2")
simics> @conf.foo.invalid_attr = configuration.INVALID()
```

5.2.4 Accessing Command Line Commands from Python

At times, it can be useful to access command line commands from a Python script file. This is done using the ***run_command(cli_string)*** function, which takes a string which is then evaluated by the command line front-end. For example, write `run_command("pregs")` to execute the **pregs** command. Any return value from the command is returned to Python. There is also the ***quiet_run_command***, which captures any output produced by the command and returns it. More details about both functions can be found in the *Simics Reference Manual*.

5.2.5 The Simics API

The Simics API is a set of functions that provide access to Simics functionality from loadable modules (i.e., devices and extensions), and Python scripts. All functions in the Simics API have a name that starts with "***SIM_***". They are described in details in the *Simics Reference Manual*.

By using the **api-help** and **api-search** commands you can get the declarations for API functions and data types. **api-help** ***identifier*** will print the declaration of ***identifier***. **api-search** ***identifier*** lists all declarations where ***identifier*** appears.

Note that while **api-help** ***topic*** does the same thing as **help api:*topic***, the **help-search** command will not search through the API declarations.

The Simics API functions are available in the `simics` Python module. This module is imported into the Python environment in the frontend when Simics starts. However, for user-written `.py` files, the module must be imported explicitly:

```
from simics import *
```

Errors in API functions are reported back to the caller using *frontend exceptions*. The exception is thrown together with a string that describes the problem more in detail. Examples of exceptions are `General`, `Memory`, `Index`, and `IOError`. In DML and C/C++, these exceptions have to be tested for using ***SIM_clear_exception*** or ***SIM_get_pending_exception***. In Python, such exceptions result in regular Python exceptions.

For the Python environment, Simics defines an exception subclass for each of its defined exceptions in the `simics` module. These are raised to indicate exceptions inside the API functions. When errors occur in the interface between Python and the underlying C API function, the standard Python exceptions are used; e.g., if the C API function requires an `int` argument, and the Python function is called with a `tuple`, a Python `TypeError` exception is raised.

5.2.6 Haps

A *hap* is an event or occurrence in Simics with some specific semantic meaning, either related to the target or to the internals of the simulator.

Examples of simulation haps are:

- Exception or interrupt
- Control register read or write
- Breakpoint on read, write, or execute
- Execution of a magic instruction (see the *Hindsight User's Guide*)
- Device access

There are also haps which are related to the simulator, e.g., (re)starting the simulation or stopping it and returning to prompt.

Note: In Simics documentation, the word *event* is used exclusively for events that occur at a specific point in simulated time, and *hap* for those that happen in response to other specific conditions (like a state change in the simulator or in the simulated machine).

The callback can be invoked for all occurrences of the hap, or for a specified range. This range can be a register number, an address, or an exception number, depending on the hap.

A complete reference of the haps available in Simics can be found in the *Simics Reference Manual*.

Example of Python Callback on a Hap

This example uses functions from the Simics API to install a callback on the hap that occurs when a control register is written. It is intended to be part of a `.simics` script, that extends an *MPC8641-Simple* machine setup. The ***SIM_hap_add_callback_index()*** function sets the index of the control register to listen to, in this case the `%dec` register in an ppce600 processor.

```
@dec_reg_no = conf.mpc8641d_simple.soc.cpu[0].iface.int_register
```

```
.get_number("dec")

# print the new value when %dec is changed
@def ctrl_write_dec(user_arg, cpu, reg, val):
    print "[%s] Write to %%dec: 0x%x" % (cpu.name, val)

# install the callback
@SIM_hap_add_callback_index("Core_Control_Register_Write",
                            ctrl_write_dec, None, dec_reg_no)
```

In CLI, the same example would look like:

```
script-branch {
    local $cpu = (current-processor)
    while TRUE {
        $cpu.wait-for-register-write dec
        echo "[" + $cpu + "] Write to %dec: " + ((hex ($cpu.read-reg dec)))
    }
}
```

Example of Python Callback on Core_Log_Message_Extended Hap

This example shows how to add a callback to the Core_Log_Message_Extended. This allows for better control when handling log messages. This example writes all log messages to a file that is associated with the **cpu** object.

Note: The Core_Log_Message_Extended hap will only be triggered for messages with log level less than or equal to the log level setting of the object.

```
from cli import *
from simics import *

class file_log(object):

    def __init__(self, file, object, level):
        # setup logging
        try:
            self.f = open(file, 'w')
        except Exception, msg:
            raise Exception("Failed to open file %s, %s" % (file, msg))
        self.object = object
        self.level = level

        # install the callback
        SIM_hap_add_callback(
```

```
            "Core_Log_Message_Extended", self.log_callback, None)

    def log_callback(self, dummy, obj, type, message, level, group):
        type_str = conf.sim.log_types[type]
        if self.object == obj and self.level <= level:
            self.f.write("[%s %s] %s,  level=%d, group=%d\n" % (
                    SIM_object_name(obj), type_str, message, level, group))

file_log('log_cpu.out', conf.ebony.soc.cpu, 1)
```

Chapter 6

Configuration and Checkpointing

Simics includes a *configuration system* used to describe the state of the simulated machines.

Simics's configuration system is object-oriented: a simulated machine is represented as a set of *objects* that interact when the simulated time advances. Typically, processors, memories and devices are modeled as objects. Each object is defined by a number of properties that are called *attributes*. A processor object, for example, will have an attribute called *freq_mhz* to define its clock frequency (in MHz).

Simics's configuration system is flexible: it is possible to create and delete objects dynamically, as well as access the attributes of all objects for reading and writing at any time.

Simics's configuration system allows its elements to be saved so that a complete simulated machine state can be written to a set of files. This set of files is called a *checkpoint*.

This chapter describes the Simics configuration system as well as the different layers built on top of it to handle more specific tasks:

- Checkpoints: how a simulated state is saved and restored via checkpointing;
- Inspection: how the simulated state can be examined and changed during simulation;
- Start Scripts: the components and scripts used to define the initial state of a machine in the examples provided with Simics.

6.1 Basics

As mentioned above, Simics's configuration system is object-oriented. A Simics object is instantiated from a Simics *class*. The core of Simics defines some useful classes, but most of the classes (processors, device models, statistic gathering extensions) are provided by *modules* that are loaded by the simulator when requested.

For example, the **x86-p4** module defines, not surprisingly, the **x86-p4** class. Note that a module may define several classes. Since modules advertise the classes they define, Simics can load modules transparently as objects are instantiated.

A class defines *attributes* that represent both the static and dynamic state of the instantiated objects. The static state includes information that does not change during the simulation (like a version number in a register) while the dynamic state covers the part of the device that are affected by the simulation (registers, internal state, buffers, etc.).

Let us take the example of an x86-p4 processor and have a closer look on how it can be configured using attributes:

- We can create an object instantiated from the class **x86-p4**. Let us call it **cpu0**
- The attribute *freq_mhz* can be set to 1500. It defines the processor clock frequency (in MHz)
- The attribute *physical_memory* can be set to a memory space object, such as **phys_mem0**. This attribute points to the object that will answer to the memory accesses coming from the processor.

As you noticed, attributes may be of various types. A complete description is available in the next section.

6.2 Checkpointing

Simics's configuration system can save the complete state of a simulation in a portable way. This functionality is known as *checkpointing*, and the set of files that represent the elements of the systems are called a *checkpoint*.

Saving and restoring a checkpoint can be done on the command line with the **write-configuration** and **read-configuration** commands.

A checkpoint consists of the following files, collected under a directory:

- A main *configuration file*, named `config`. This is a text representation of the objects present in the system.
- Optional image files (described in section 6.2.3), named after each respective image object.

Below is a portion of a checkpoint file showing an object. Saved objects are always represented as `OBJECT` *object-name* `TYPE` *class-name* `{` *attributes* `}`. In this case we have an instance of the **DEC21143** class (fast Ethernet LAN controller interfacing the PCI bus) named **dec0**. The attribute *pci_bus* connect the device to the PCI bus.

```
OBJECT dec0 TYPE DEC21143 {
        queue: cpu0
        component: eth_adapter_cmp0
        component_slot: "dec"
        object_id: "dec0"
        build_id: 0xbb9
        pci_bus: pci_bus0
        ...
}
OBJECT ... TYPE ... {
...
```

Objects are saved in the main checkpoint file in no specific order.

6.2.1 Compatibility

Simics maintains checkpoint compatibility with older versions, i.e. it is always possible to continue using checkpoints created in a previous version of Simics when upgrading to a new version. Compatibility is always maintained for one major version older than the oldest of the supported API versions. For checkpoints older than that, load the checkpoint with a newer version of Simics it and create a new checkpoint.

The opposite is not true. Trying to load a checkpoint created in a newer version of Simics than the local version will typically not work. The same restriction may apply even between minor Simics releases. For example, a checkpoint created with Simics 3.2.2 is not guaranteed to load correctly in the older Simics 3.2.1 release.

6.2.2 Attributes

The short example of the **dec0** description only uses a few types of attribute values: strings, objects, and hexadecimal integers. The possible attribute types are:

string
Strings are enclosed in double quotes, with C-style control characters: `"a string\n"`

integer
Integers can be in hexadecimal (`0xfce2`) or signed decimal (`-17`) notation.

boolean
One of `TRUE` and `FALSE`.

floating-point
Specified in decimal (`1.0e-2`) or hexadecimal (`0x5.a21p-32`) style, just like in C.

object
The name of a configuration object: `cpu0`.

list
Comma-separated list of any attribute values, enclosed in parentheses. Example: `("a string", 4711, (1, 2, 3), cpu0)`

dictionary
The format is a comma-separated list of key/value pairs, like in: `{ "master-cpu" : cpu0, "slave-cpu" : cpu1 }`. The key should be a string, integer or object, while the value can be of any attribute type. Dictionaries are typically used to save Python dictionaries in a checkpoint. Keys must be unique, although Simics does not enforce this.

Each attribute belongs to one of the following categories. Note that only attributes of the first two categories are saved in checkpoints.

Required
Required attributes must be set when creating an object. They are saved in checkpoints. If you edit a checkpoint, you should never remove a required attribute—Simics will complain and refuse to load the checkpoint if you do.

Optional
If no other value is provided, optional attributes take their default value when the object is created. They are saved in checkpoints, but if you edit them out they will revert to their default value when the checkpoint is loaded.

Session
Session attributes are only valid during a Simics session. They are not saved in checkpoints. They are usually used for statistics gathering or values that can be computed from the rest of the object state.

Pseudo
Pseudo attributes are not saved in checkpoints and usually contain read-only information that does not change, or that is calculated when the attribute is accessed. Pseudo attributes are in some cases used to trigger state changes in the object when written.

6.2.3 Images

Simics implements a special class called **image** for objects that potentially need to save a huge amount of state, like memories and disks. An image represents a big amount of raw data using pages and compression to minimize disk usage.

To save space and time, images do not save their entire state every time a checkpoint is written. They can work in two ways:

- Images can save their state *incrementally*. At each checkpoint, an image saves the difference between its current state and the previously saved state (either the previous checkpoint or the initial state). This is the default behavior implemented by Simics. This allows several checkpoints to be saved and restored using the same base image and a series of difference files.
- Images can be used as *read-write* media. In that case the file representing the data is always up to date to the current state. However, this prevents the image from being used in a previously saved checkpoint or initial state, since its contents are modified as the simulation advances.

It is important to understand that when used in incremental mode, images create *dependencies* between checkpoints. A checkpoint can only be loaded if all previous checkpoints are intact.

An example, let us have a look at an assumed disk image:

```
    ...
}
OBJECT disk0_image TYPE image {
        ...
        files: (("tango1-fedora5.craff", "ro", 0, 0x4c5abc000, 0),
                ("disk0_image.craff", "ro", 0, 0x4c5abc000, 0))
        ...
}
```

...

The checkpointed image is based on the file `tango1-fedora5.craff`, on top of which is added the file `disk0_image.craff` that contains the difference between the checkpoint and the initial state.

Files like `disk0_image.craff` are often called *diff files* because they contain the difference between the new state and the previous state.

Image Search Path

This section contains more in-depth explanations about image handling that you may skip when reading this guide for the first time.

When successive checkpoints are saved, an image object may become dependent on several diff files present in different directories. To keep track of all files, Simics stores in the checkpoint a *checkpoint path* list that contains the absolute directory paths where image files may be found. Images file names are then saved as `%`*n*`%\`*filename* where `%`*n*`%` represents the number of the entry in the checkpoint path, counting from zero.

Note: Simics's checkpoint path is different from Simics's search path (see section 4.4), although both will be used when looking for image files, as show below.

To summarize, when loading a checkpoint or a new configuration, Simics looks for images in the following way:

- If the filename does not contain any path information (like `image.craff`) or contains a relative path (like `test\image.craff`), the file is looked up *first* from the checkpoint directory, *then* from all the path entries in Simics's search path, *in order* (see also section 4.4 for more information).

  Windows example;
  If Simics's search path contains `[workspace]\targets\mpc8641-simple\` and the checkpoint is located in `C:\checkpoints`, Simics will look for the file `test\image.craff` in the following places:

  1. `C:\checkpoints\test\image.craff`
  2. `[workspace]\targets\mpc8641-simple\test\image.craff`

  Unix example;
  If Simics's search path contains `[workspace]/targets/mpc8641-simple/` and the checkpoint is located in `/home/joe/checkpoints/`, Simics will look for the file `test/image.craff` in the following places:

  1. `/home/joe/checkpoints/test/image.craff`
  2. `[workspace]/targets/mpc8641-simple/test/image.craff`

- If the filename contains a checkpoint path marker (`%`*n*`%`), the marker is translated using Simics's checkpoint path and the file is looked up in the corresponding path.

Windows example;
If Simics's checkpoint path contains `C:\checkpoints\c1;C:\checkpoints\c2`, the file `%1%/image.craff` will be translated into `C:\checkpoints\c2\image.craff`.

Unix example;
If Simics's checkpoint path contains `/home/joe/c1:/home/joe/c2`, the file `%1%/image.craff` will be translated into `/home/joe/c2/image.craff`.

- If the filename contains an absolute path the file path is used as is.

  Windows example;
  `C:\checkpoints\image.craff`

  Unix example;
  `/home/joe/image.craff`

Note: The reason why Simics's search path is involved in the process is that it makes writing new configurations easier. Adding a path to the place where all initial images are located allows you to just specify the image names.

6.2.4 Saving and Restoring Persistent Data

As an alternative to checkpointing, Simics allows you to only save the *persistent* state of a machine, i.e., data that survive when the machine is powered-down. This typically consists of disk images and flash memory or NVRAM contents. A persistent data checkpoint is handled exactly like any other checkpoint and contains the same file set, but only objects containing persistent data are saved. This persistent data checkpoint can be loaded on top of a corresponding configuration later on.

The commands **save-persistent-state** and **load-persistent-state** respectively save and load the persistent data in a configuration.

Note: These commands are often used to save the state and reboot a machine after the disk contents have been modified. Remember that the target OS **might have cached disk contents in memory**. In order to have a clean disk that can be used at boot, you should synchronize the disk, for example by running `init 0` on a Unix target system, or shutting down the operating system, before you issue the **save-persistent-state** command.

6.2.5 Modifying Checkpoints

Checkpoints are usually created by saving a configuration inside Simics, but it is possible to edit or even create checkpoints yourself. It may even be required to edit file paths in a checkpoint file if it is relocated.

Because a minimal checkpoint only has to include required attributes, *creating a checkpoint from scratch* works relatively well for small configurations. We suggest you use an existing

checkpoint as a starting point if you wish to do that. Note that more advanced layers have been built on top of the configuration system to make the creation of a new machine much easier. Refer to section 6.6 for more information.

Modifying checkpoints require some extra care. Adding or removing devices may confuse the operating system, which does not expect devices to appear or disappear while the system is running, and cause it to crash.

Changing the processor frequency may be enough to confuse the operating system. Many operating systems check the CPU frequency at boot time, and base their waiting loops and timing on the value they got. Saving a checkpoint and changing the frequency after boot may affect the simulation and confuse the system. Devices that use processor frequency to trigger events at specific times may also behave strangely when the frequency suddenly changes.

6.2.6 Merging Checkpoints

If you want to make a checkpoint independent from all previous checkpoints, for example to distribute it, you can use the small **checkpoint-merge** program in `[simics]\bin` from your system command line. It merges the checkpoint with all its ancestors to create a checkpoint that has no dependencies. Specify the checkpoint you want to distribute as the first parameter and the name of the new stand-alone checkpoint as the second. This tool can be used in both Unix and Windows environments.

6.3 Inspecting the Configuration

Object attributes that are of type `integer`, `string` or `object` are directly accessible at the *command line* with the notation *object−>attribute*:

```
# reading the EAX register in an x86 processor
simics> cpu0->eax
0
# writing a new value to EAX
simics> cpu0->eax = 10
simics> cpu0->eax
10
simics>
```

More information about the command line and scripting is available in chapter 5.

Finally, objects and attributes (of all types) are also available when *scripting Simics directly in Python*. Configuration objects are available under the `conf` namespace:

```
# reading the EAX register in an x86 processor
simics> @conf.cpu0.eax
0
# writing a new value to EAX
simics> @conf.cpu0.eax = 10
simics> @conf.cpu0.eax
```

```
10
simics>
```

More information about scripting Simics in Python is available in chapter 5.

6.4 Components

All machines in `[simics]\targets\`*architecture* use components to create configurations. A component is typically the smallest hardware unit that can be used when configuring a real machine, and examples include motherboards, PCI cards, hard disks, and backplanes. Components are usually implemented in Simics using several configuration objects and can also contain subcomponents.

Components are intended to reduce the large configuration space provided by Simics's objects and attributes, by only allowing combinations that match real hardware. This greatly simplifies the creation of different systems by catching many misconfigurations.

Components themselves are also configuration objects in Simics. But to avoid confusion, they will always be referred to as components and the objects implementing the actual functionality will be called objects.

6.4.1 Component Definitions

The *component* is the basic building block in the component system. When a component is created, it is in a *non-instantiated* state. At this stage only the component itself exists, not the configuration objects that will implement the actual functionality. Once a complete configuration has been created, all included components can be *instantiated*. When this happens, all objects are created and their attributes are set.

Components are connected to each other in with *connectors*. Each connector has a *connector type* which tells what kind of connector it is and a *direction*, which can be *up*, *down*, or *any*. A connector is either required or optional. If it is optional it does not need to be connected to anything. Unless a connector is specified as *hotpluggable* it cannot be connected or disconnected after the component is instantiated. If a connection is hotpluggable it must be optional.

Connectors can be connected to each other in *connections*. Each connection connects an up connector with a down connector. A connection can also include an any connector. If an any connector is connected to an up connector it works exactly like a down connector and if it is connected to a down connector it works exactly like an up connector. The connections in the system must not form a cycle. You can think of the components and connections in the system as a directed acyclic graph with the components as the nodes and the connections as the arcs.

Each connected subgraph in the set of components is called a *component hierarchy*.

A component is said to be *above* another component if it can be reached through up connectors in one or more steps from that component. A component A is said to be *below* a component B if B is above A.

A *root* is a component without any components above it. A component's roots are the roots which are above it.

A component where the *top_level* attribute returns true is a *top-level* component. It is often a motherboard, backplane or system chassi. It must be a root.

A *standalone* component is a component without any required connectors. A typical example is a hotplug device, such as a PC Card (PCMCIA) or an Ethernet link.

To instantiate a set of components each component which is not standalone or top-level must have a top-level component as a root.

Components are also *namespaces* and can be nested in a *namespace hierarchy*, which is separate from the component hierarchy. The root of the hierarchy is the *global namespace*, and this is the only namespace which is not a component. Each configuration object (including components) lives in a namespace. The object is a *child* of the namespace and the namespace is the *parent* of the object. The other objects in the namespace are *siblings* of the object.

6.4.2 Importing Component Commands

Components in Simics are grouped by machine architecture, or by type, into several modules. Before a component can be used in Simics, the corresponding component module has to be loaded. When the component module is loaded, CLI commands for creating components are added to the front end. The most common modules, that are not architecture specific, are `memory`, `pci`, `std` and `timing`. To import all modules that are used by the *MPC8641-Simple* machine for example, issue the following commands:

```
simics> load-module std-components
simics> load-module memory-components
simics> load-module mpc8641-comp
```

6.4.3 Creating Components

The **create-**<**component**> command is used to create non-instantiated components. There is one create command for each component class. The arguments to the create command represent attributes in the component. Standalone components can be created both non-instantiated and instantiated. To create instantiated components, there are **new-** commands, similar to the **create-** commands.

The following code creates a non-instantiated 'board-mpc8641-simple' component representing an *MPC8641-Simple*, called 'mpc8641_simple0'

```
simics> load-module mpc8641-comp
simics> create-board-mpc8641-simple cpu_frequency = 133.33
Created non-instantiated 'board_mpc8641_simple' component 'mpc8641_simple0'
```

6.4.4 Connectors

A connector provides a means for a component to connect to other components. Connectors have a defined direction: *up*, *down*, or *any*. The direction is *up* if it needs an existing hierarchy to connect to; for example, the PCI-bus connector in a PCI device must connect to a PCI slot.

A connector has a *down* direction if it extends the hierarchy downwards; for example, a PCI slot is a connection downward from a board to a PCI device. There are also non-directed connectors, with direction *any*. You can only connect an *up* to a *down* connector or to an *any* connector, and similar for *down* connectors. Connectors with the *any* direction can not be connected together.

Many connectors have to be connected before the component is instantiated, while others can be empty. A standalone component, as described above, may have all connectors empty.

A *hotplug* connector supports connect and disconnect after instantiation. Other connectors can only be connected, or left unconnected, when the configuration is created and may not be modified after that point. A *multi* connector supports connections to several other connectors. Creating *multi* connectors should be avoided, it is often better to dynamically create non-*mutli* connectors when new connectors are needed.

It is not possible to connect instantiated components with non-instantiated ones. The reason is that the instantiated component expects the other to have all objects already created, and need to access some of them to finish the connection.

The **info** command of a component lists all connectors and some information about them:

```
simics> mpc8641_simple0.info
Information about mpc8641_simple0 [class board_mpc8641_simple]
============================================================

Slots:
     ddr_slot[0] : mpc8641_simple0.ddr_slot[0]
     ddr_slot[1] : mpc8641_simple0.ddr_slot[1]
     ddr_slot[2] : mpc8641_simple0.ddr_slot[2]
     ddr_slot[3] : mpc8641_simple0.ddr_slot[3]
          eth[0] : mpc8641_simple0.eth[0]
          eth[1] : mpc8641_simple0.eth[1]
          eth[2] : mpc8641_simple0.eth[2]
          eth[3] : mpc8641_simple0.eth[3]
          mac[0] : mpc8641_simple0.mac[0]
          mac[1] : mpc8641_simple0.mac[1]
          mac[2] : mpc8641_simple0.mac[2]
          mac[3] : mpc8641_simple0.mac[3]
       pci_slot1 : mpc8641_simple0.pci_slot1
       pci_slot2 : mpc8641_simple0.pci_slot2
    rapidio_port : mpc8641_simple0.rapidio_port
       serial[0] : mpc8641_simple0.serial[0]
       serial[1] : mpc8641_simple0.serial[1]
             soc : mpc8641_simple0.soc

Connectors:
     ddr_slot[0] : mem-bus              down
     ddr_slot[1] : mem-bus              down
     ddr_slot[2] : mem-bus              down
```

```
 ddr_slot[3] : mem-bus              down
      eth[0] : ethernet-link        down  hotplug
      eth[1] : ethernet-link        down  hotplug
      eth[2] : ethernet-link        down  hotplug
      eth[3] : ethernet-link        down  hotplug
      mac[0] : phy                  up
      mac[1] : phy                  up
      mac[2] : phy                  up
      mac[3] : phy                  up
   pci_slot1 : pci-bus              down  hotplug
   pci_slot2 : pci-bus              down  hotplug
rapidio_port : rapidio              down  hotplug
   serial[0] : serial               down  hotplug
   serial[1] : serial               down  hotplug
```

The MPC8641 board has four slots for DDR SDRAM modules, one PCI slot, one PCIe slot, two serial ports, and four Ethernet ports. The memory slot is not listed as *hotplug* since DDR modules have to be inserted when the machine is configured initially, while serial and Ethernet ports support connect and disconnect at run-time.

To enable input and output for the simulated machine, the following commands create a serial text console and connects it to the `serial[0]` connector of the MPC8641 board.

```
simics> load-module std-components
simics> connect mpc8641_simple0.serial[0] (create-std-text-console).connector_serial
```

In this example the std-text-console connector is called `connector_serial` although its real name is `serial`. This is because the std-text-console is an old-style component that has not been updated to the more recent way of writing components. As a result, the connector object has to be given instead of the connector. With new components, the two are the same thing.

Since the machine needs some memory to run, we also add a DDR memory module to our example. A CLI variable is used to hold the name of the memory component.

```
simics> load-module memory-components
simics> $ddr = (create-ddr-memory-module rank_density = 128 module_data_width = 64)
simics> connect mpc8641_simple0.ddr_slot[0] $ddr.connector_mem_bus
```

6.4.5 Instantiation

When a component hierarchy has been created, it can be instantiated using the **instantiate-components** command. This command will look for all non-instantiated top-level components and instantiate all components below them. The **instantiate-components** command can also be given a specific component as argument. Then only that component will be instantiated, including its hierarchy if it is a top-level component.

```
simics> instantiate-components
```

If there are unconnected connectors left that may not be empty, the command will return with an error.

When the instantiation is ready, all object and attributes have been created and initialized. In our example, a text console window should have opened. The hardware of the simulated MPC8641 board is now properly configured, but since no software is loaded, it will not show any output on the console if the machine is started.

6.4.6 Inspecting Component Configurations

The **list-components** command prints a list of all components in the system. All connectors are included, and information about existing connections between them.

The **info** namespace command provides static information about a component, such as the slots and a list of connectors.

The **status** namespace command provides dynamic information about a component, such as attribute values and a list of all current connections. The output from status in the MPC8641 example:

```
simics> mpc8641_simple0.status
Status of mpc8641_simple0 [class board_mpc8641_simple]
=====================================================

Setup:
    Top component : mpc8641_simple0
     Instantiated : True
      System Info :

Attributes:
        cpu_cores : 2
    cpu_frequency : 133.33
        lm_offset : False
      pcie2_agent : False
         rtc_time : 2008-06-05 23:52:01 UTC

Connections:
      ddr_slot[0] : ddr_memory_cmp0:mem_bus
        serial[0] : text_console_cmp0:serial
```

6.4.7 Accessing Objects from Components

When doing more advanced configuration of a machine, it may be necessary to access configuration objects and their attributes directly. Each object in a component has slot name that can be used for accessing the object. A list of slot names, and their mapping to actual

configuration object names, is available in the output from the component's **info** command. The next example prints the *pc* attribute from the *cpu* object.

```
simics> p -x mpc8641_simple0.soc.cpu[0]->pc
```

Getting an object for non-instantiated component does not work since they do not have any associated configuration objects. But it is possible to access the `pre_conf_objects` of the non-instantiated component from Python in the same way. The following example works both for instantiated and non-instantiated components:

```
simics> @print "SVR = 0x%x" % conf.mpc8641_simple0.soc.cpu[0].svr
'SVR = 0x80900100'
```

Remember that not all configuration object attributes are available on a `pre_conf_object`. Only attributes that have been assigned by the component during initialization exists.

6.4.8 Available Components

The *Target Guide* for each architecture lists and describes all components that are available.

6.5 Object Name

An object can be identified using more than one name. This section describes the different ways of identifying an object.

All objects have a name that is used when printing log messages, writing checkpoints, in CLI commands, etc. The ***SIM_object_name*** function returns this object name. This name will be refereed to as the *object name* in this section, even though an object can have several names for identification.

The object name is the name the object is given when created or the objects location in the hierarchy. The name an object gets is controlled by the preference variable *legacy_name*. Section 6.5.1 describes the legacy mode when the objects name is its given name. Section 6.5.2 describes the mode when the object name is identical to its hierarchical location. The *legacy_name* variable's default value is `false`.

An object can be given a name when created. The ***SIM_create_object***, ***SIM_add_configuration***, or ***SIM_set_configuration*** functions takes the object name as argument. The given name can be a string without dots "foo", a string with dots "cmp0.foo", an empty string "", or None. Later sections describes how the given name affects the object name.

An object's *hierarchical location* is defined by its *component* and *component_slot* attributes. The hierarchical location for an object is the name of the *component* that the object's *component* attribute points at, and the component's *component_slot* attribute string, the two concatenated with a dot. For example, an object that belongs to a component named "cmp0" with the slot name "bar" has the hierarchical location "cmp0.bar".

All objects that reside in a slot in a component have valid *component* and *component_slot* attributes. It is the component's responsibility that the attributes are valid. The attributes are set when an object is added to a slot. A name that contains dots is a hierarchical location.

All objects also have an ID. The ***SIM_object_id*** function returns the object ID as a string. The object ID is unique and is never changed and will be saved in checkpoints. The object ID will not change even if the object is moved around in the hierarchy or is given a new name. The object ID is sometimes identical to the object's given name.

The object name and object ID are always unique. Creating an object and giving it a name that already exist will generate an error.

If the given name is a hierarchical location an object will be added to that hierarchical location even if the *component* and *component_slot* attributes are not set. Simics will extract the component name and slot name from the given name. This information is then used for looking up the component and adding the object to the slot via the `component` interface. An object given the "cmp0.cmp1.foo" belongs to the component "cmp0.cmp1" and has the slot name "foo". Note that the component "cmp1" in this example belongs to the component "cmp0".

6.5.1 Legacy Name Enabled

Legacy names are enabled when the *legacy_name* preference variable is `true`. In this mode an object name is the given name when created. The default value is `false`.

An object given a name without dots "foo" gets the name "foo" and the object ID "foo".

An object given a name with dots "cmp0.foo" gets the name "cmp0.foo" and a generated object ID of the form "obj_XYZ", as the object ID never can be a hierarchical location, i.e. a name with dots.

An object given an empty name "" or None will get a name of the form "obj_XYZ" and the same as its object ID.

Given Name	Name	ID	Name After Move
foo	foo	foo	foo
cmp0.foo	cmp0.foo	obj_XYZ	cmp0.foo
None	obj_XYZ	obj_XYZ	obj_XYZ

The *given name* is the name that the user has provided. *Name* is the name the object gets when created. *ID* is the unique object ID. *Name After Move* is the name the object gets after being moved to *cmp1* to the slot named *smurf*. Notice that an object's name is never changed, compare with the next section.

6.5.2 Legacy Name Disabled

Legacy names are disabled when the *legacy_name* preference variable is `false`. In this mode an object name is the hierarchical location of the object.

An object given a name without dots "foo" will get a name that is the hierarchical location of the object if the *component* and *component_slot* attributes are valid. Otherwise it will get the name "foo", which means that the object does not reside in any slot in a component. The object given the name "foo" will get the object ID "foo" independent of the *component* and *component_slot* attributes.

An object given a name with dots "cmp0.foo" gets the name "cmp0.foo" or, if it is put in the slot "bar", the name "cmp0.bar", its hierarchical location, and an object ID of the form "obj_XYZ".

An object given an empty name "" or None will get a hierarchical location if the *component* and *component_slot* attributes are valid, otherwise a name of the form "obj_XYZ". The object ID will always be of the form "obj_XYZ", identical to the object name if *component* and *component_slot* attributes are invalid.

Given Name	Name (Slot = None)	Name (Slot = bar)	ID	Name After Move
foo	foo	cmp0.bar	foo	cmp1.smurf
cmp0.foo	cmp0.foo	cmp0.bar	obj_XYZ	cmp1.smurf
None	obj_XYZ	cmp0.bar	obj_XYZ	cmp1.smurf

The *given name* is the name that the user has provided. *Name* is the name the object gets when created, depending on if *component* and *component_slot* are set: *Slot = None* when they are not set; *Slot = bar* when they are set and *component_slot* is *bar*. *ID* is the unique object ID. *Name After Move* is the name the object gets after being moved to *cmp1* to a slot named *smurf*.

6.5.3 Looking Up Object

The ***SIM_get_object*** function takes *name* as a string argument. *name* can be an object identifier, name, object ID, or hierarchical location. Even an object with a non hierarchical name, when *legacy_name* is `true`, can be looked up using its hierarchical location.

When calling ***SIM_get_object***, the order in which a name is looked up is: object ID, object name, hierarchical location. Note that object ID, object name, and hierarchical location can be the same or different.

6.6 Ready-to-run Configurations

Simics includes many customizable ready-to-run configurations. Because checkpoint files are by definition very static, these example configurations are not checkpoint-based, but rather build on *components* and *scripts* to generate a working simulated machine.

The example configurations are located in separate directories for each system *architecture*: `[simics]\targets\`*architecture*. Each of these directories contains a number of Simics scripts (i.e., files containing Simics commands):

<machine>-common.simics
: Script that defines a complete simulated machine, i.e., both hardware and software, that can be run by Simics directly. The `common` script uses the `-system.include` script to define the hardware, and the `-setup.include` script for software configuration. The `-common.simics` scripts may add additional hardware in some cases.

  These are the files you want to use to start the standard example machines in this directory.

  <machine> in the script name is either a Unix machine name, or a some other name that defines the hardware/software combination.

<machine>-<feature>-common.simics
: A script that extends the `-common.simics` script with a new feature. Many minor features, such as the processor frequency, can be controlled using parameters to the `common` script, but features that are mutually exclusive are added as separate scripts. Typical examples are scripts that add different diff-files to the same disk image in the system setup.

<architecture-variant>-system.include
: Script that defines the hardware of a machine. This script can be shared by several simulated machines that are based on the same hardware. The hardware setup is typically configurable using some standard parameters.

<machine>-setup.include
: Script that defines the software and possibly configures the machine to run the selected software, for example setting boot path, and scripting automatic login.

The example configurations are designed to work with the disk images distributed with Simics. The machines are described in the *Target Guide* corresponding to each architecture.

Several machines may be defined for a given architecture, and thus the corresponding architecture directory will contain several *machine*`-common.simics` scripts.

6.6.1 Customizing the Configurations

There are several ways to customize the examples provided with Simics. They are listed below ordered by how simple they are to use.

Parameters
: The machine scripts distributed with Simics can be modified by setting parameters (CLI variables) before the script is actually run. This is the easiest way to change the default configuration. Parameters can typically be used to change properties such as the amount of memory, the number of processors and the primary MAC address. The available parameters are listed in each *Target Guide*.

  Setting parameters before the script is run can be done in two fashions:

  1. Use the startup flag `-e`. The flag must be repeated for each parameter to set. An example on Windows:

     ```
     $ simics.bat -e '$freq_mhz = 133.33' -e '$host_name = "target0"' ↵
               targets/mpc8641-simple/mpc8641d-simple-linux.simics
     ```

     An example on Linux:

     ```
     $ ./simics -e '$freq_mhz = 133.33' -e '$host_name = "target0"' ↵
               targets/mpc8641-simple/mpc8641d-simple-linux.simics
     ```

2. Launch Simics without any script, set the parameters at CLI, and run the CLI command **run-command-file**.

   An example identical to the one above:

```
$ simics.bat

simics> $freq_mhz = 100
simics> $host_name = "target0"
simics> run-command-file targets/mpc8641-simple/mpc8641d-simple-linux.simics
```

Scripts

A simulated machine is defined by several scripts, as described above. By replacing the `-common.simics` file with a user defined script, the system can be configured more in detail while keeping the machine definition provided by the `-system.include` file. Similarly the `-setup.include` can be replaced to configure different software on the same machine.

Components

Components represents real hardware items such as PCI cards, motherboards, and disks. Using components to configure a machine provides freedom to set up the simulated machine in any way that is supported by the architecture. The `-system.include` files use components to create their machine definitions. A complete description of components is provided earlier in this chapter.

Objects and Attributes

A component is implemented by one or more configuration objects in Simics, and each object has several attributes describing it. Configuring machines on the object and attribute level is *not* supported in Simics, and such configurations may not work in future versions.

Below is another example of a simple configuration based on *MPC8641-Simple*, that uses parameters to configure two machines slightly differently that both run in the same Simics session.

```
$freq_mhz = 133.33
$host_name = "target0"
run-command-file "%script%/mpc8641d-simple-linux.simics"

$freq_mhz = 200.00
$host_name = "target1"
run-command-file "%script%/mpc8641d-simple-linux.simics"
```

6.6.2 Adding Devices to Existing Configurations

The parameters available for each predefined machine allows the user to do minor modifications. It is also possible to extend the ready-to-run configurations with additional components without creating new machine setups from scratch.

Since the machine setup scripts are located in the read-only master installation of Simics, they should not be modified. User files that add new components should instead be placed in the corresponding `[workspace]\targets\`*architecture* directory.

Here is a short example of how to extend the *MPC8641-Simple* to add a SCSI disk:

```
run-command-file "%script%/mpc8641d-simple-linux.simics"

if (not defined scsi_disk_image) or (not defined scsi_disk_size) {
    interrupt-script ("This script requires that the image and size of a "
                      + "bootable scsi disk has been specified.")
}

run-command-file "%script%/mpc8641-simple-system.include"

$sym = (create-pci-sym53c810)
$scsi_bus = (create-std-scsi-bus)
$scsi_disk = (create-std-scsi-disk scsi_id = 0
                                   size = $scsi_disk_size
                                   file = $scsi_disk_image)

$system.connect pci_slot1 $sym
$scsi_bus.connect $sym
$scsi_bus.connect $scsi_disk

instantiate-components
```

Notice, that script requires you to provide a disk image and a valid disk size to be possible to run. Essentially the script will run another script, `mpc8641-simple-linux.simics`, which will create an instantiated *MPC8641-Simple* machine, then a SCSI disk is created and connected to that machine, and finally the new disk is instantiated.

It is possible to add many devices to an instantiated Simics machine in a similar manner. In the case a device must be added to the target machine before instantiation, write a script as described above.

Chapter 7

Moving Data in and out of the Simulation

In order to use Simics, you must have *images* (also called *disk dumps*) with the operating system and the applications you plan to run. Depending on the type of machine you are using, these images will corresponds to the contents of a disk, a flash memory, a CD-ROM, etc. There are some images provided that work with the example machines located in the `targets` directory.

Simics images are usually stored in a special format called `craff` (for Compressed Random Access File Format) to save disk space. Simics also accepts a raw binary dump as an image. This is sometimes more practical if you are manipulating images outside Simics. Simics comes with the `craff` utility to manipulate and convert images in `craff` format (see section 7.1.7).

This chapter will explain the following:

- How to work with images in general
- How to use CD-ROMs and floppies with Simics
- How to use the SimicsFS file system
- How to import the contents of a real disk inside Simics

To provide you with a more practical overview, here are the ways you can install and modify the operating system and the applications you wish to run:

Using Simics:

- You can install a completely new OS or simply copy files using a simulated CD-ROM drive, by linking it to a real CD-ROM drive on your host machine (Linux only) or by using a CD image file (refer to sections 7.2.1 and 7.2.2).
- You can copy files from the simulated floppy drive by linking it to the real host floppy device (Linux only) or by using a floppy image file (see sections 7.2.3 and 7.2.4).

- You can use SimicsFS to directly access your host file systems from the simulated machine (see section 7.4).
- You can download files over the simulated network (see the *Ethernet Networking User's Guide*).

Do not forget to read more about *images* in section 7.1.1 to learn how to save and re-use your changes.

Using External Programs

- You can install a new OS along with new programs on a real machine and create an image from the real machine storage (disk, flash memory, etc.). Section 7.6 shows how to perform this with a disk.
- You can modify a FAT image directly with Mtools (see section 7.1.4). (Linux only)
- You can modify an image directly via a loopback device (see section 7.1.5). (Linux only)
- Use a tool such as SFMount from PassMark ® Software. (Windows only)

7.1 Working with Images

7.1.1 Saving Changes to an Image

If you modify or create new files on a storage device within Simics, you should remember that by default images are *read-only*. This means that the alterations made when running Simics are *not* written to the image, and will last only for the current Simics session. As described in the 6.2.3 section, this behavior has many advantages. You may however want to save your changes to the image to re-use them in future simulations.

The first thing you should do is to make sure that all the changes are actually written to the media you are using. In many cases, the simulated operating system caches operations to disks or floppies. A safe way to ensure that all changes are written back is to shutdown the simulated machine.

When all changes have been written to the media *in* the simulation, you can save the new contents of the image in different ways:

- Using the **save-persistent-state** command, all image changes for persistent storage media are saved to disk as a persistent state. **This is the recommended way of saving your image changes**.
- Using the ⟨**image**⟩**.save-diff-file** command, you can manually save a diff file for the images you are interested in.
- Using the ⟨**image**⟩**.save** command, you can create a raw binary dump of the image that is completely independent of all previous images and diff files. Note that this is **not** the best way to get a new and shiny image, since the image is saved uncompressed, which can take a lot of time and disk space.

Note: The ⟨**image**⟩**.save** actually allows you to save a *partial* dump of an image, which may be useful to dump a specific part of a disk or a floppy.

Once you have saved the images, you can do the following:

- If you used **save-persistent-state**, you can issue the **load-persistent-state** command just after starting the original configuration. This will add the new changes to the persistent storage media images and the machine will boot with the changes included. **This is the recommended way of using a saved persistent state**.

  For example, let us suppose that you saved some new files on the disk of the *enterprise* machine (started with the `enterprise-common.simics` script). You saved the persistent state of the machine after stopping it to the persistent state file `new-files-added`. You can easily create a small script to start *enterprise* with the new files:

  ```
  # enterprise-new-files.simics
  run-command-file enterprise-common.simics
  load-persistent-state new-files-added
  ```

- You can also load the original configuration and add the diff files manually to the images, using the ⟨**image**⟩**.add-diff-file** command.

- If you are building your own configurations (either as scripts or as checkpoints), you can add the diff files to the *files* attribute of the corresponding **image** object. This corresponds to what the ⟨**image**⟩**.add-diff-file** command does.

If you save several persistent states or image diff files that are dependent on each other, it may become cumbersome to take care of all these dependencies and to remember which files are important or not. You can *merge* the states or image diff files to create a new independent state:

- If you are working with persistent states, you can use the `checkpoint-merge` utility to create a persistent state that is independent of all previous files, including the original images provided with Simics. **This is the recommended way of creating a new independent image**. You can load it as usual with the **load-persistent-state** command.

- If you saved some image diff files manually, you can use the `craff` utility described below to merge the diff files yourself.

7.1.2 Reducing Memory Usage Due to Images

Although images are divided into pages that are only loaded on request, Simics can run out of host memory if very big images are used, or if the footprint of the software running on the simulated system is bigger than the host memory. To prevent these kind of problems,

Simics implements a global image memory limitation controlled by the **set-memory-limit** command.

When Simics is started a default memory-limit is automatically set based on the amount of physical memory available on the host and if Simics runs in 32 or 64 bit mode. The default memory-limit does not consider if other applications and users are running on the same host, nor what kind of target system that is simulated in Simics. (For example each target processor will allocate additional non-image memory, so for systems with many processors the default limit could be too high).

When the memory limit is reached, Simics will start swapping out pages to disk very much like an operating system would do. The **set-memory-limit** command let you specify the maximum amount of memory that can be used, and where swapping should occur.

Note: This memory limitation only applies to *images*. Although this is unlikely, Simics can run out of memory due to other data structures becoming too large (for example memory profiling information) even though a memory limit has been set.

7.1.3 Using Read/Write Images

As mentioned in section 6.2.3, images can also work as read-write media, although this is **not** recommended. It can be useful sometimes when planning to make large changes to an image (like installing an operating system on a disk).

To make an image read-write in your own configurations, simply set the second parameter (the "read-only" flag) of the *files* attribute in the image object to `"rw"`.

In a ready-to-run example like *enterprise*, you can change this attribute after the configuration is completed:

```
# read the 'files' attribute
simics> @files = conf.disk0_image.files
simics> @files
[['enterprise3-rh73.craff', 'ro', 0, 20496236544L, 0]]
# provide the complete path to the file
simics> @files[-1][0] = "[workspace]/targets/enterprise/images/
enterprise3-rh73.craff"
# change the second element to make the file read-write
simics> @files[-1][1] = "rw"
# check the result
simics> @conf.disk0_image.files
[['[workspace]/targets/enterprise/images/enterprise3-rh73.craff',
 'rw', 0, 20496236544L, 0]]
```

Note that by indexing `files` with the index `-1`, the last element of the array is accessed, which is always the one that should be set read-write, in case `files` is a list of several files.

Simics does not look for files in the Simics search path when the files are used in read-write mode. If you do not provide a complete path to a read-write file, a new file will be created in the current directory.

The image file must be a raw image file. Simics does not support using craff files in read/write mode.

Use this feature with caution. Make sure to take a copy of the original image before running Simics with the image in read-write mode. Remember to synchronize the storage device within the target OS before exiting Simics, for example by shutting down the simulated machine.

7.1.4 Editing FAT Images Using Mtools

This is a Linux specific chapter. If you have an image that contains a FAT file system, you can use Mtools (`http://mtools.linux.lu`) to get read-write access to the image. You must have a raw binary dump of the image for Mtools to work. This can be obtained using the `craff` utility (see section 7.1.7).

A few wrapper scripts around Mtools are included in the Simics distribution in the scripts directory.

create-fat.sh
Creates an image with a formatted FAT file system.

ls-fat.sh
Lists files in an image.

copy-to-fat.sh
Copies one or more files or directories to an image.

copy-from-fat.sh
Copies a file or directory from an image.

If your image is partitioned (a complete disk for example), you may need to give Mtools special parameters like an *offset* or a *partition*. Please see the Mtools documentation for more information.

7.1.5 Editing Images Using Loopback Mounting

This is a Linux specific chapter. If the host OS supports loopback devices, like, e.g., Linux, you can mount an image on your host machine and get direct read/write access to the files within the image. If you have root permissions this allows you to easily and quickly copy files.

Note: Remember that the image must be a raw binary dump. Disk dumps supplied with Simics are normally in `craff` format but you can use the `craff` utility to unpack the disk image to a raw image. The resulting images have the same size as the simulated disk, so you need to have sufficient free space on your host disk to contain the entire simulated disk image.

Note: Do not try to loopback mount an image over NFS. This does not work reliably on all operating systems (Linux, for example). Instead, move the image to a local disk and mount it from there.

On Linux:

```
mount <disk_dump> mnt_pnt -o loop=/dev/loopn,offset=m
```

Example:

```
# mount /disk1/rh6.2-kde-ws /mnt/loop -o loop=/dev/loop0,offset=17063424
# cd /mnt/loop
# ls
bin   dev  home  lost+found  opt   root  tmp  var
boot  etc  lib   mnt         proc  sbin  usr
#
```

As shown in the example, the disk dump containing a Red Hat 6.2 KDE WS is mounted on the `/mnt/loop` directory. The file system mounted on `/` starts on the offset 17063424 on the disk. Linux autodetects the file system type when mounting (ext2 in this example). If you want to access another kind of file system, use the *-t* fs option to the mount command. Once the file system is mounted, you can copy files in and out of the disk image.

The `offset` can be calculated by examining the partition table with `fdisk` (from within Simics). Use `mount` to find the partition you want to edit or examine (e.g., `/dev/hda2` is mounted on `/usr` which you want to modify). Next, run `fdisk` on the device handling this partition (such as `fdisk /dev/hda`). From within `fdisk`, change the display unit to sectors instead of cylinders with the **u** command and print the partition table with **p**. You will now see the start and end sectors of the partitions; you can get the offset by taking the start sector multiplied with the sector size (512).

When you have finished examining or modifying the disk, unmount it and touch the disk image. For example:

```
cd
umount /mnt/loop
touch /disk1/rh6.2-kde-ws
```

The modification date of the disk image does not change when you modify the disk via the loopback device. Thus, if you have run Simics on the disk image earlier, the OS might have cached disk pages from the now modified disk image in RAM. This would cause a new Simics session to still use the old disk pages instead of the newly modified pages. Touching the image file should ensure that the OS rereads each page.

7.1.6 Constructing a Disk from Multiple Files

In some cases, you may want to populate a simulated disk from multiple files covering different parts of the disk. For example, the partition table and boot sectors could be stored

in a different disk image file than the main contents of the disk. If that is the case, you cannot use the ⟨**image**⟩**.add-diff-file** command: you must set manually the disk image *files* attribute to put each image file at its appropriate location.

Assume you are simulating a PC and want to build a disk from a main file called `hda1_partition.img` and a master boot record image file called `MBR.img`. The main partition will start at offset 32256 of the disk, and the MBR (Master Boot Record) covers the first 512 bytes of the disk (typically, you would get the contents of these image files from the real disk as detailed in section 7.6). The following command in Simics's start-up script will build the disk from these two files.

```
load-module std-components
create-std-ide-disk disk2 size = 2559836160
@image = get_component_object(conf.disk2, 'hd_image')
@image.files = [["hda1_partition.img", "ro", 32256, 1032151040, 0],
                ["MBR.img",            "ro",     0,        512, 0]]
```

Note that the two image files cover non-overlapping and non-contiguous sections of the disk.

7.1.7 The Craff Utility

The images distributed with Simics, and in general most of the images created by Simics, are in the `craff` file format. The `craff` utility can convert files to and from the `craff` format, and also merge several `craff` files into a single file.

In your Simics distribution you will find `craff` in **Windows:** `[simics]\bin`, **Linux:** `[simics]/bin`. The examples below assume that `craff` is present in your shell path.

Convert a raw dump to `craff` format

```
> craff -o mydisk.craff mydisk.img
```

Convert a single `craff` file to a raw file

```
> craff --decompress -o mydisk.img mydisk.craff
```

Merge multiple craff files into a single `craff` file

If more than one input file is specified, they will be merged so that later files override earlier files on the command line. The input craff files in this example come from several checkpoints.

```
> craff -o merged.craff chkpt1.craff chkpt2.craff chkpt3.craff
```

Add a `craff` file to a raw dump, producing a new dump

```
> craff --decompress -o new.img mydisk.img diff.craff
```

The input files can be any combination of raw and `craff` files.

Make a file of the differences of two dumps

```
> craff --diff -o diff.craff dump1.img dump2.img
```

The resulting file, `diff.craff`, will contain only what is needed to add to `dump1.img` in order to get `dump2.img`. This is useful to save space if little has been changed.

See also the *Simics Reference Manual* for a full description of the `craff` utility and its parameters.

7.2 CD-ROMs and Floppies

7.2.1 Accessing a Host CD-ROM Drive

This is a Linux specific chapter. Accessing the CD-ROM of the host machine from inside the simulation is supported on Linux hosts. This is done by creating a **file-cdrom** object using the **new-file-cdrom** command. First, you should insert the CD in the host machine and figure out which device name it uses.

On a Linux host, this is typically `/dev/cdrom`, which is a symbolic link to the actual CD-ROM device, e.g., `/dev/hdc`. Note that you need read/write access to the CD-ROM device for this to work.

When you have the correct device file name, you create a **file-cdrom** object and insert it into the simulated CD-ROM drive:

```
simics> new-file-cdrom /dev/cdrom file-cd0
cdrom 'file-cd0' created
simics> cd0.insert file-cd0
Inserting media 'file-cd0' into CDROM drive
```

Note that you must replace `/dev/cdrom` with the correct host device name as mentioned above, and **cd0** with the correct Simics object name. Use the **list-objects** command to find the correct object of class **scsi-cdrom** or **ide-cdrom**.

The **cd0.insert** command simulates inserting a new disk into the CD-ROM drive, and there is also a corresponding **cd0.eject** command that simulates ejecting the disk.

7.2.2 Accessing a CD-ROM Image File

A file containing an ISO-9660 image can be used as medium in the simulated CD-ROM. This image file can be created from real CD-ROM disks, or from collections of files on any disk.

On Linux, an image can be created from a set of files with the `mkisofs` program. For example:

`mkisofs -l -L -o` *image* `-r` *dir*

On Windows, you can use a third-party product to create ISO-9660 images from files or from CD-ROMs, and a non-exhaustive list is given in figure 7.1. Note that many programs can read CD-ROMs in either “file” or “raw” mode (“raw” mode is often called “aspi”). If CD-ROMs are read using file mode, the resulting image will not be bootable.

Once you have an image file, a **file-cdrom** object can be created, and then inserted into a simulated CD-ROM device in the same way as above:

```
simics> new-file-cdrom myimage.iso
cdrom 'myimage' created
```

WinImage	`http://www.winimage.com` Shareware, only copies images from real CD-ROM
WinISO	`http://www.winiso.com` Shareware
UltraISO	`http://www.ezbsystems.com/` Shareware
MagicISO	`http://www.magiciso.com` Shareware, can make images from files, CD-ROMs, and DVD-ROMs, and edit ISO images.
mkisofs	`ftp://ftp.berlios.de/pub/cdrecord/alpha/win32/` Part of the cdrtools package, free, need Cygwin

Figure 7.1: Windows Programs to Create ISO-9660 Images

```
simics> cd0.insert myimage
Inserting media 'myimage' into CDROM drive
```

Note that **cd0** above refers to the Simics object name of the CD-ROM drive. This may, or may not be called **cd0**. To see which object name to use, try the **list-objects** command and look for an object of class **scsi-cdrom** or **ide-cdrom**.

7.2.3 Accessing a Host Floppy Drive

This is a Linux specific chapter. It is possible to access a floppy on the host machine from within Simics if the host is running Linux. For example (assuming the floppy device object is called **flp0**):

```
simics> flp0.insert-floppy A /dev/fd0
```

Note: To boot directly from the floppy on a simulated x86 architecture you need to select the "A" drive to be the boot device (in, for example, `enterprise-common.simics`):

```
simics> system_cmp0.cmos-boot-dev A
```

7.2.4 Accessing a Floppy Image File

Sometimes it can be convenient to have copies of boot floppies as image files. On Windows, to create an image of a floppy you can use for example WinImage (see section 7.2.2 above).

On Linux, you can use the `dd` command:

```
dd if=/dev/fd0 of=floppy.img
```

It is then possible to use this image file in Simics:

```
simics> flp0.insert-floppy A floppy.img
```

Note: To boot directly from the floppy on a simulated x86 architecture you need to select the "A" drive to be the boot device (in, for example, `enterprise-common.simics`):

```
simics> system_cmp0.cmos-boot-dev A
```

Floppies are also a convenient way to move small amounts of data out of the simulated machine. Write the data to the simulated floppy inside the simulated machine, and then extract it from the image.

If it is formatted as FAT file system, a floppy image can be manipulated with Mtools (Linux only, see section 7.1.4 for more information).

7.3 USB disks

Virtual USB disks can be used to transfer files to and from virtual machines. Virtual USB disks store their data in images files using the age old FAT file system. See section 7.1.4 for more information about how you can access FAT file system images.

A virtual USB disk is created with the **new-usb-disk-from-image** command. The command accepts a raw FAT file system image and creates all the necessary objects and initializes the partition table on the virtual device.

```
simics> $usb_disk=(new-usb-disk-from-image /tmp/stick.img)
```

Next, you can insert the virtual USB disk into the machine. Exactly how that is done depends on the system that is being simulation. It should be connected to an empty USB-port connector. This is an example of how that is done in a x86 Intel ® 440BX AGPset:

```
simics> south_bridge_cmp0.connect usb1 $usb_disk
```

Depending on how the USB host controller is implemented, you may also need to instruct it to start performing USB list traversal. This is how that is done in a x86 Intel ® 440BX AGPset:

```
simics> (south_bridge_cmp0.get-component-object pci_to_usb) ↵
->frame_list_polling_enabled = 1
```

Writes to the virtual USB disk are normally not written directly to the image file, but rather cached in memory. The save command can be used to write a new image that will include all sectors written by the virtual machine.

```
simics> ($usb_disk.get-component-object scsi_disk_image).save new_image ↵
0x100000
```

The image is mapped at an offset of 1 MiB (0x100000 bytes), meaning that starting the save at that offset will create a clean image file just like the one that was originally fed into **new-usb-disk-from-image**. The first 1 MiB is used for the partition table which does not need to be mapped.

7.4 Using SimicsFS

SimicsFS gives you access to the file system of your real computer inside the simulated machine. This greatly simplifies the process of importing files into the simulated machine.

SimicsFS is supported for targets running Solaris 7, 8, 9 and 10, and Linux kernel versions 2.0, 2.2, 2.4 and 2.6.

SimicsFS is installed on disk dumps distributed with Simics. For users booting from other disks, there are a number of steps needed to configure the target system. This process is target OS specific, and is described in the following sections.

SimicsFS is not fully functional on all simulated operating systems. The following limitations apply:

Simulated OS	Limitations
Linux	Access is read-only. (Write support experimental.)
Solaris	Truncating files does not work.
Other	SimicsFS is not currently available.

7.4.1 Installing SimicsFS on a Simulated Linux System

Since there are several different Linux kernels, and a module has to match the kernel version, there are only pre-compiled `simicsfs` modules provided with Simics as parts of complete operating system images. If you run your own version of Linux you have to compile SimicsFS yourself. To build SimicsFS, first download the source for SimicsFS from `https://www.simics.net/pub/`. There are two ways to install SimicsFS with your Linux System. You can either add it to the source tree of your Linux kernel or compile it as an external module. Below both methods are described. The description assumes that you know the basics of compiling your own Linux kernel. The most up to date documentation, however, is provided with SimicsFS itself.

Note: The exact file name for the SimicsFS download depends on the version of SimicsFS. Below we use `simicsfs.tar.gz`, but you should use the name of the package you downloaded instead.

When the instructions ask you to copy files into the simulated machine, one of the methods described elsewhere must be used (e.g., network, loopback disk access, or CD-ROM).

Do the following to add SimicsFS to a 2.6 kernel:

- Unpack the `simicsfs.tar.gz` file in the `linux-2.6.x/fs/` directory.
- Rename `Makefile_2.6` to `Makefile`.
- Add the following lines to `fs/Makefile` and `fs/Kconfig`:

linux-2.6.x/fs/Makefile

```
obj-$(CONFIG_SIMICSFS) += simicsfs/
```

linux-2.6.x/fs/Kconfig

```
config SIMICSFS
        tristate "Simics hostfs"
        help
          This is filesystem to be used by simics to acces the host filesystem

          To compile this file system support as a module, choose M here: the
          module will be called simicsfs.

          If unsure, say N.
```

- Configure the Linux build, select to compile SimicsFS either as a module or as a part of the kernel.
- Compile (and install) the kernel. If you compile SimicsFS as a module you must also build and install its modules.

Do the following to add SimicsFS to a 2.4 kernel:

- Unpack the `simicsfs.tar.gz` file in the `linux-2.4.x/fs/` directory.
- Rename `Makefile_2.4` to `Makefile`.
- Add the following lines to `fs/Makefile` and `fs/Kconfig`:

linux-2.4.x/fs/Makefile

```
obj-$(CONFIG_SIMICSFS) += simicsfs/
```

linux-2.4.x/fs/Kconfig

```
tristate 'Simics hostfs' CONFIG_SIMICSFS
```

- Configure the Linux build, select to compile SimicsFS either as a module or as a part of the kernel.
- Compile (and install) the kernel. If you compile SimicsFS as a module you must also build and install its modules.

Do the following to install SimicsFS for a running 2.6 kernel:

- Unpack the `simicsfs.tar.gz` file.
- Make sure that the kernel source/devel package is installed.
- Rename Makefile_2.6 to Makefile.
- Change to the `simicsfs` directory, and compile the module:

```
$ make -C /lib/modules/`uname -r`/build M=`pwd`
```

- Create a new directory `/lib/modules/`*version*`/kernel/fs/simicsfs/` on the simulated machine, where *version* is the simulated machine's kernel version.
- Copy the newly compiled SimicsFS module file to the directory `/lib/modules/`*version*`/kernel/fs/simicsfs/` on the simulated machine, and make sure it is called `simicsfs.o` for Linux 2.4 and `simicsfs.ko` for Linux 2.6.

The SimicsFS driver communicates with Simics through a virtual device mapped into the memory space of the simulated system. The address the device is mapped at varies from system to system based on where there is a convenient slot in the systems memory map. The driver looks for the device at the most common address for the processor architecture, but for Linux 2.6 you can also configure the address where the driver looks manually. You do this differently depending on whether you load SimicsFS as a module or if you have compiled it into the kernel statically.

When SimicsFS is compiled as a module you configure it by setting the `phys_addr` parameter to the physical address of the SimicsFS device:

`insmod` *simicsfs-module-filename* `phys_addr=`*address*

When SimicsFS is compiled as a static part of the kernel you have to configure the physical address by setting the `simicsfs` parameter on the kernel command line. This can either be compiled into the kernel or passed from the bootloader, check the documentation for the target software you use for the details.

When you have installed the module do the following to configure Linux to use it:

- Create the mount point on the simulated machine with `mkdir /host`.
- Add the following line in the simulated machine's `/etc/fstab` (replace `/host` with your mount point):

```
special      /host      simicsfs     noauto,ro      0 0
```

Note: To use the experimental write support, change `ro` into `rw`.

- Mount SimicsFS with the command `mount /host` on the simulated machine.

SimicsFS should now be working, and by issuing `ls /host` on the simulated machine, you should get a listing of the host machine's files.

Installing SimicsFS Pseudo Device for Linux

The SimicsFS functionality is implemented in two parts: a device driver that is installed in the simulated operating system and a special device, or *pseudo device* that is installed in the simulated hardware.

The special device is of the Simics class **hostfs**, and it needs to be mapped in on the same physical address as the device driver is configured to use.

While many of the virtual platforms provided with Simics already are configured to include the **hostfs** device, you may have to add it yourself. To see if your system already includes the device, you can do the following:

```
simics> list-objects type = hostfs
  :
Class     Object
----------------
<hostfs>  hfs0
simics> devs hfs0
Count    Device  Space  Range                    Func/Port
      0  hfs0    plb    0xff660000 - 0xff66000f 0
simics> cpu0->physical_memory
"plb"
```

This shows that there already exists a **hostfs** device called **hfs0**. It is mapped into the **plb** memory space, which is the physical memory space for the CPU **cpu0**.

If you did not find any **hostfs** device, you have to add it yourself. Exactly how to do that differs between different virtual platforms, depending on how their CPUs address physical memory. In most cases, something like this will work:

```
simics> $hostfs = python "SIM_create_object('hostfs', 'hfs0', [])"
simics> (cpu0->physical_memory).add-map $hostfs 0xff80_0000 16
Mapped 'hfs0' in 'plb' at address 0xff800000.
```

Note that the **hostfs** device needs to be mapped into 16 consecutive bytes, and you must make sure that there is nothing else mapped on the same addresses. An empty space in physical memory can be found by running **(cpu0->physical_memory).map**.

7.4.2 Installing SimicsFS on a Simulated Solaris System

These are the steps needed to install SimicsFS on a simulated Solaris, version 7, 8, 9 and 10. When the instructions ask you to copy files into the simulated machine, one of the methods described above must be used (network, loopback disk access, CD-ROM...). Note that the driver included with earlier Simics distributions was called `hostfs` and not `simicsfs`.

- Create a new directory `/usr/lib/fs/simicsfs/` on the simulated machine.
- Copy the file `[simics]/import/sun4u/mount_simicsfs` (**Linux**), `[simics]\import\sun4u\mount_simicsfs` (**Windows**) to `/usr/lib/fs/simicsfs/` on the simulated disk, and rename it to `mount`.
- Copy the file `[simics]/import/sun4u/simicsfs-sol`*version* (**Linux**), `[simics]\import\sun4u\simicsfs-sol`*version* (**Windows**) (where *version* matches the version of Solaris running on your simulated machine) to `/usr/kernel/fs/sparcv9/` on the simulated machine, and rename it to `simicsfs`.
- Add the following line to `/etc/vfstab` on the simulated disk:

```
simicsfs  -       /host   simicsfs  -       no      -
```

- Create the mount point on the simulated machine with `mkdir /host`.
- When the simulated system is running, issue the command:

```
mount /host
```

You should now be able to do `ls /host` on the simulated system to get a list of the files on the host.

7.4.3 Using SimicsFS

By default, the simulated machine can access the entire file tree (**Linux**: `/`, **Windows:** `C:\`) of the host computer from the mount point (typically `/host`).

This can sometimes be inconvenient (or dangerous, if the simulator runs untrusted or unreliable code), so it is recommended to set the directory that is visible to the simulated machine using the ⟨**hostfs**⟩**.root** command, e.g.:

Linux:

```
simics> hfs0.root /home/alice/sandbox
```

Windows:

```
simics> hfs0.root d:\misc\sandbox
```

(This is also a way to access drives other than `C:`.)

The command will take effect next time SimicsFS is mounted.

Because of implementation limitations, it is recommended that SimicsFS be chiefly used to copy files into and out from the target machine. In particular, executing binaries residing on the host machine may be unreliable.

Note: When saving a checkpoint while a SimicsFS is mounted, take care that the host files that were used at that time are kept unchanged when the checkpoint is loaded.

7.5 Using TFTP

If you have Simics Ethernet Networking, it is possible to transfer files from the host environment to the target (simulated) machine by using the TFTP feature provided by the **service-node**. Since TFTP executes in lockstep, with only one packet acknowledged at a time, it is slower than for example FTP, but it reliably transports files between the host and target machines.

It is assumed that a service node has been created and connected to the Ethernet device through an Ethernet link. The following example presents how TFTP is used on a target machine which is running Linux and has the `tftp` program installed. Furthermore, the target machine in this example is using the IP address 10.10.0.10 and the service node uses 10.10.0.1.

Creating a service node and connecting it to the target machine can be done with the **connect-real-network** command. This will also set up port forwarding to the real network, even though this is not a requirement for using TFTP:

```
simics> connect-real-network 10.10.0.10
```

After booting the target machine into Linux the first step is to bring up the network interface that is connected to the service node. At the target prompt, issue:

```
joe@computer: ~# ifconfig eth0 10.10.0.10 up
```

To transfer the file `myfile.txt` from the host machine, issue:

```
joe@computer: ~# tftp -l myfile.txt -g 10.10.0.1
```

The directory that the service node uses to find files downloaded by the target can be changed with the ⟨**service-node**⟩**.set-tftp-directory** command. This also controls where uploaded files are saved. The default is to search the Simics path, starting with the current working directory of the Simics process. The search path can be changed with the **add-directory** command and can be viewed with the **list-directories** command.

7.6 Importing a Real Disk into Simics

It is possible to create an image by copying data from a real disk. If the disk to be copied contains an operating system, you must have at least two operating systems on the machine, since the partition that should be copied should **not** be in use or mounted.

Before making a copy of a disk, some information about the disk should be gathered:

- The number of disk cylinders
- The number of sectors per track
- The number of disk heads
- The offset where the specific partition starts (optional)

- The size of a specific partition (optional)

On Linux, these numbers can be obtained using the `fdisk` utility. You can choose to make a copy of the whole disk or just a partition from the disk using the `dd` utility. Example:

```
dd if=/dev/hdb of=hdb_disk.img
```

On Windows, You can use the System Information application to find the information under `Components/Storage/Disks`. You have to select the **Advanced** setting from the **View** menu. If you have the Cygwin tool set (`http://www.cygwin.com`) installed, you can use the `dd` utility to create the image, provided that the correct entries in the `/dev` file system are created. To access the first hard drive (`/dev/hda`):

```
mkdir -p /dev/hda
mount -s -b '\\.\PHYSICALDRIVE0' /dev/hda
```

You can also mount a specific drive letter:

```
mkdir -p /dev/fd0
mount -s -b '\\.\A:' /dev/fd0
```

or

```
mkdir -p /dev/hda1
mount -s -b '\\.\C:' /dev/hda1
```

Cygwin's `mount` program creates persistent mounts (they are stored in the registry), so you will only need to set these things up once. The *-b* option to `mount` ensures that no CR/LF conversions are made. See the Cygwin documentation for further details on how to use the `mount` command.

On Windows hosts without Cygwin, a third-party program can be used to create the disk images. See figure 7.1 for more details.

Note: To save space, you may want to compress the disk image using the `craff` utility. See section 7.1.7.

The next step is to prepare the target configuration so it can use the new disk. For x86 targets, the *dredd* machine has a `$disk_files` parameter that can be set to a list of files to use in the image object of the boot disk, and also `$disk_size` that specifies the size of that disk.

```
$disk_size = 1056964608
$disk_files = [["hdb_disk.img", "ro", 0, 1056964608, 0]]
```

For other machines, that do not have these parameters, attributes in the disk object and its corresponding image objects have to be set instead.

Make sure to set the `$disk_size` correctly to reflect the size of the disk that has been copied. If only a partition has been copied, the offset where the partition starts, and the size of the partition, should be set in the file list. If the whole disk has been copied, the offset is zero and the size should be the size of the whole disk. Several partitions can be combined to form the complete disk, as described in section 7.1.6.

For an x86 machine, the system component will automatically set the BIOS geometry for the `C:` disk. It can also be set manually:

```
simics> system_cmp0.cmos-hd C 1023 16 63
```

Chapter 8

Serial Links

8.1 Serial Link Component

Connecting simulated machines over a simulated serial connection is done by creating a **ser-link** component that connects to the serial devices in the machines. The link object can be thought of as modeling a serial cable that is plugged into the connectors on the devices—and just like a real cable, it is a point-to-point connection that connects exactly two devices.

The link object models serial communication at the character level in a simplified way. The bandwidth for the connection is configured in the link object, which means that the serial devices do not need to be explicitly configured by software.

New **ser-link** components can be added with the **new-ser-link** command:

```
simics> load-module ser-link
simics> new-ser-link
Created instantiated 'ser_link' component 'serial_link0'
```

Serial connectors of other components can then connect to that link. The serial link has two connectors, **device0** and **device1**, representing the two endpoints of the cable. For an *MPC8641-Simple* machine, the second UART can be connected to the link the following way:

```
simics> connect serial_link0.device0 mpc8641d_simple.serial1
```

8.1.1 Text and Telnet Consoles

In addition to simulated serial devices, either endpoint of a serial link can be hooked up to a text console or a telnet console. (And just as with serial devices, you have the option of connecting the two endpoints directly without having a link in between—though this is generally much more useful with consoles.)

A text console will open a terminal window on the host computer, and lets the user talk to the connected serial device. A telnet console is similar, except that instead of opening a terminal window, it starts a telnet server; the user can then use any telnet program to connect to this server, and talk to the connected serial device.

You create text and telnet consoles by instantiating **std-text-console** and **std-telnet-console** components, and connecting them to the link or device you want them to talk to.

8.2 Host Serial Console

Host Serial Console, which is a part of Simics Hindsight, is a way of connecting a terminal application through a serial port on the host machine to a serial device object in Simics. The procedure is almost identical in both Linux and Windows. The following examples show how to create and connect a **host-serial-console** to a *MPC8641-Simple* machine, first in Linux:

```
simics> mpc8641d_simple.disconnect "serial[0]"
Disconnected mpc8641d_simple from mpc8641d_simple.console0
simics> mpc8641d_simple.connect "serial[0]" ↵
    (new-std-host-serial-console port = /dev/pts/1)
[host_serial_console_cmp0.con info] pseudo device opened: /dev/pts/1
```

And the same procedure in Windows:

```
simics> mpc8641d_simple.disconnect "serial[0]"
Disconnected mpc8641d_simple from mpc8641d_simple.console0
simics> mpc8641d_simple.connect "serial[0]" ↵
    (new-std-host-serial-console port = COM1)
[host_serial_console_cmp0.con info] COM port opened: COM1
```

The host-serial-console will operate at the baud rate and other attributes that are set for the physical serial port of the host. In Linux these host settings are edited with **stty**. In Windows these settings must be edited from a **Command Prompt**, this is an example:

```
C:\> mode com1: baud=4800 parity=n data=8 stop=1
```

It is also possible to connect the host-serial-console to a virtual serial port. In Linux this is called pseudo-terminal or pseudo-device and if there is no port specified for the host-serial-console any free `pts` is opened. The terminal application connects to the opened pseudo device.

However, in Windows a virtual serial port must have been created in advance. There exist several Windows third-party utilities which create virtual serial port pairs for various purposes. In the following example, first such a pair has been created with the port names

COM98 and COM99, then host-serial-console connects to port 98, and finally the terminal application can connect to port 99 and the two can communicate over the pair:

```
simics> mpc8641d_simple.disconnect "serial[0]"
Disconnected mpc8641d_simple from mpc8641d_simple.console0
simics> mpc8641d_simple.connect "serial[0]" ↵
    (new-std-host-serial-console port = COM98)
[host_serial_console_cmp0.con info] COM port opened: COM98
```

The above examples leave the original console disconnected. To set up a *MPC8641-Simple* machine with a host-serial-console but without the superfluous console window we edit the start scripts. The following lines in the machine start script creates a *MPC8641-Simple* connected to a host-serial-console connected to Windows COM1:

```
$system = (create-board-mpc8641-simple ...
$console = (create-std-host-serial-console port=COM1)
$system.connect serial0 $console
```

8.2.1 Connecting to the Host Serial Console using the Windows HyperTerminal

Once the host-serial-console has been configured within Simics it is possible to connect to it using a standard terminal program running on your host system. In this section we will use the HyperTerminal application, which comes bundled with some Windows versions, to connect to the simulated system.

First launch HyperTerminal from **Start Menu** → **All Programs** → **Accessories** → **Communications** → **HyperTerminal**. The dialog *Connection Description* will appear, as shown in Figure 8.1. Name the Connection "Simics" and press *OK*. In the next dialog, select the COM-port that Simics has opened in the field *Connect using*. In the next dialog, just accept the default settings and press *OK*.

Now, resume the simulation in Simics. The output from the simulated serial console will appear in the HyperTerminal window. See figure 8.2.

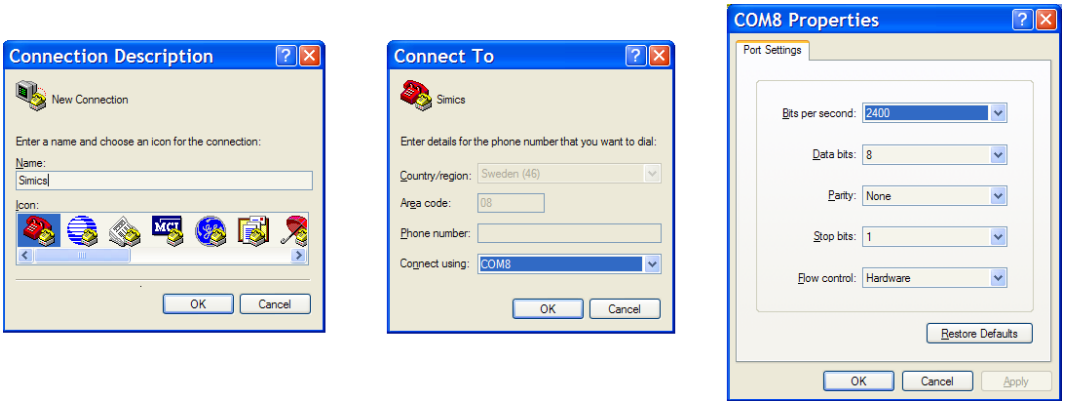

Figure 8.1: Connecting to Simics using the Windows HyperTerminal

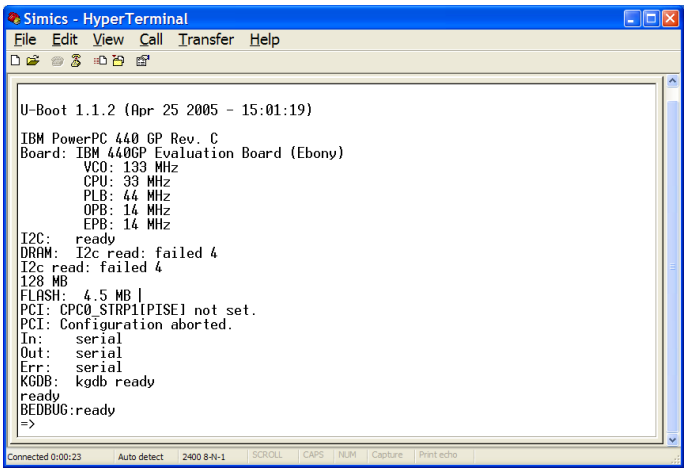

Figure 8.2: HyperTerminal connected to a simulation

Part III

Software Debugging

Chapter 9

Introduction

This part of the document explains how to use Simics Hindsight as a tool in software development. In this setting, Simics plays two roles:

execution platform
: The software is run on a simulated target machine inside Simics.

debugger
: Simics can either function as a stand-alone debugger, or as a debugger backend for an external debugger.

Using Simics as a debugger has some major benefits compared to debugging on real hardware:

- Debugging is completely non-intrusive.
- You can inspect and modify the state of the entire target system, at any level. This is especially convenient when debugging low-level code such as firmware and hardware drivers.
- The simulation is completely deterministic. Once you manage to trigger a bug, you can repeat it as often as you like.
- You have complete control over time. For example, you can freeze time while inspecting the target, save the simulated state in a checkpoint and restore it at a later time, and even run the whole simulation backward.

Note: It is recommended that you have read *Getting Started* before tackling this document.

Chapter 10

Debugging Software with Simics

This chapter explains two cornerstones of debugging in Simics: its powerful breakpoint support, and fully scriptable debug and symbol information handling. It also discusses the reverse execution capabilities of Simics.

Note: Much of the breakpoint support is available even if you use Simics as a backend for an external debugger (as in chapter 12 and 14), but most of the symbolic debugging features are not exposed through these interfaces. This is not a big deal in practice, since these debuggers have their own symbolic debugging support.

10.1 Breakpoints

Like any other debugger, Simics can set breakpoints on user code and data. But unlike most debuggers, Simics breakpoints are not limited by what the hardware can support; for example, there is no restriction on the number of read/write breakpoints (also known as watchpoints).

In Simics you can set breakpoints on:

- memory accesses: any range and combination of read/write/execute
- time (number of cycles or instructions executed)
- instruction types, such as control register accesses
- device accesses
- output in the console

Simics is fully deterministic, and debugging in Simics is fully non-intrusive. This makes it possible to narrow down the location of difficult bugs by re-running the *exact* same run as many times as you need.

But there is often no need to run by the bug multiple times, since breakpoints work even when the simulation is running backwards. For example, to find the code responsible for writing garbage to a pointer, run forward until your program crashes, then set a write

breakpoint on the (now clobbered) pointer, and run backward; the breakpoint will trigger at the point in time when the pointer was last written to.

10.1.1 Memory Breakpoints

A memory breakpoint stops the simulation whenever a memory location in a specified address interval is accessed. The address interval can be of arbitrary length and the type of the memory access can be specified as any combination of *read*, *write*, and *execute*.

The easiest way to set memory breakpoints is to use the **break** command:

```
simics> break p:0x10000
Breakpoint 1 set on address 0x10000 in 'mpc8641d_simple.soc.phys_mem' with
access mode 'x'
```

Prefix the address with `p:` or `v:` to get a physical or virtual address, respectively. As you can see in the following example, Simics defaults to interpreting a breakpoint address as virtual if you do not specify otherwise:

```
simics> break v:0x4711
Breakpoint 2 set on address 0x4711 in 'mpc8641d_simple.cell0_context' with
access mode 'x'
simics> break p:0x4711
Breakpoint 3 set on address 0x4711 in 'mpc8641d_simple.soc.phys_mem' with
access mode 'x'
simics> break 0x4711
Breakpoint 4 set on address 0x4711 in 'mpc8641d_simple.cell0_context' with
access mode 'x'
Note: overlaps with breakpoint 2
```

This way of setting breakpoints will attach them to the memory space (physical address) or context (virtual address) connected to the current processor. If this is not exactly what you want, read on.

Note: The current processor can be set with the **pselect** command:

```
simics> pselect mpc8641d_simple.soc.cpu[0]
```

Without an argument, **pselect** prints the current processor:

```
simics> pselect
Currently selected processor is mpc8641d_simple.soc.cpu[0]
```

Physical memory breakpoints are handled by memory space objects. A memory space represents a physical address space; they sit between the processor and the actual hardware

devices (e.g. RAM) that can be accessed with read and write instructions. Breakpoints are created with the memory space's **break** command:

```
simics> mpc8641d_simple.soc.cpu[0]->physical_memory
"mpc8641d_simple.soc.phys_mem"
simics> mpc8641d_simple.soc.phys_mem.break address = 0x10000 length = 16 -w
Breakpoint 1 set on address 0x10000 in 'mpc8641d_simple.soc.phys_mem',
length 16 with access mode 'w'
```

Virtual memory breakpoints are handled by context objects. A context represents a virtual address space; you can learn more about them in chapter 10.2. Essentially, they provide a level of indirection between processors and virtual memory breakpoints; a processor has a current context, which in turn has virtual breakpoints:

```
simics> mpc8641d_simple.soc.cpu[0]->current_context
"mpc8641d_simple.cell0_context"
simics> mpc8641d_simple.cell0_context.break 0x1ff00
Breakpoint 2 set on address 0x1ff00 in 'mpc8641d_simple.cell0_context' with
access mode 'x'
```

Note that by default, all simulated processors in a cell share one context (**cell*n*_context**). If you want a virtual breakpoint to apply only to a subset of the processors, create a new context just for them:

```
simics> new-context foo
simics> mpc8641d_simple.soc.cpu[0].set-context foo
simics> foo.break 0xffffffffbfc008b8
```

Execution breakpoints can be modified with filter rules to only trigger when instructions match certain syntactical criteria. This feature is mainly useful with breakpoints covering large areas of memory. The commands available are **set-prefix** (to match the start of an instruction), **set-substr** (to match a particular substring), and **set-pattern** (to match the bit pattern of the instruction). The commands work by modifying an existing breakpoint, so you first have to set an execution breakpoint and then modify it to match only particular expressions.

For example, to stop when an instruction with the name `add` is executed in a memory range from 0x10000 to 0x12000, use the following commands:

```
simics> break 0x10000 0x2000 -x
Breakpoint 1 set on address 0x10000 in 'mpc8641d_simple.cell0_context',
length 8192 with access mode 'x'
simics> set-prefix 1 "add"
```

Simics will now break on the first add instruction encountered (or the last, if the simulation runs backward). For more information, see the *Simics Reference Manual* or use the **help break** command.

10.1.2 Temporal Breakpoints

Unlike an ordinary debugger, Simics can handle temporal breakpoints, i.e., breakpoints in time. Since the concept of time is based on steps and cycles, a temporal breakpoint refers to a specific step or a cycle count as measured by a given processor:

```
simics> cpu0.cycle-break 100
simics> cpu0.step-break 100
```

In the example above, the breakpoints are specified relative to the current time. It is also possible to give temporal breakpoints in absolute time (where 0 refers to the time when the original configuration was set up in Simics). You may set breakpoints in the past as well as in the future.

```
simics> mpc8641d_simple.soc.cpu[0].cycle-break-absolute 100
simics> mpc8641d_simple.soc.cpu[0].step-break-absolute 100
```

All the commands **cycle-break**, **step-break**, **cycle-break-absolute**, and **step-break-absolute**, can be given without prefixing them with the CPU. This will set a breakpoint for the current processor.

10.1.3 Control Register Breakpoints

A control register breakpoint is triggered when a selected control register is accessed. The control register is specified either by name or number, and the access type can be any combination of *read* and *write*. For example:

```
simics> break-cr register = msr
```

Note that the exact arguments to this command depend on the target architecture. A list of available control registers can be obtained by tab-completing the *register* argument. See the documentation for **break-cr** in the *Simics Reference Manual* for more information.

10.1.4 I/O Breakpoints

An I/O breakpoint is always connected to a specific device object. The breakpoint is triggered when that device is accessed. The breakpoint is set using the **break-io** command, which take the device name as a parameter. For example, to break on accesses to a device called **hme0**, we would use the following syntax:

```
simics> break-io device = mpc8641d_simple.soc.tsec[0]
```

A list of devices can be obtained by tab-completing the *device* argument.

10.1.5 Text Output Breakpoints

Many simulated machines have a *text console*—a terminal window hooked up to a serial port on the target machine, so that you can type commands to the target and get replies.

A text console can halt the simulation on the occurrence of a given character sequence in the output; this is called a *text output breakpoint*.

- To set a breakpoint, use the command ***console*.break *string***. Simics will stop when *string* appears in the output.
- Use ***console*.unbreak *string*** to remove a particular breakpoint.
- All breakpoints can be listed using the ***console*.list-break-strings** command.

Note: To find out if a specific simulated machine uses a text console, look for an object of class **text-console** in the list provided by **list-objects** once the configuration is loaded.

10.1.6 Graphics Breakpoints

If your target machine has a graphical display (as opposed to just a text console), you can set graphical breakpoints on it. A graphical breakpoint is a (small or large) bitmap image and a pair of coordinates; when the pixels at those coordinates on the simulated display exactly match the breakpoint image, the simulation will halt.

The following commands can be used to save and set breakpoints for a graphics console:

***gfx-console*.save-break-xy *filename left top right bottom* [*comment*]**
Let the user specify a rectangular area inside the graphics console using the top left and bottom right corners coordinates. The selected area will be saved as a binary graphical breakpoint file. You can add an optional comment that will be put at the beginning of the breakpoint file.

gfx-console*.break *filename
Activate a previously saved breakpoint and return a breakpoint id. When a graphical breakpoint is reached, it is immediately deleted and Simics halts execution and returns to the command prompt.

gfx-console*.delete *id
Delete the breakpoint associated with *id*.

10.1.7 Magic Instructions and Magic Breakpoints

For each simulated processor architecture, a special nop (no-operation) instruction has been chosen to be a **magic instruction** for the simulator. When the simulator executes such

an instruction, it triggers a `Core_Magic_Instruction` hap and calls all the callbacks functions registered on this hap.

Since magic instructions are just no-operation instructions on hardware, you can run code containing magic instructions on hardware as well as in the simulator, but you will not get any of the extra behavior Simics implements for the magic instruction.

If the architecture makes it possible, an immediate value is encoded in the magic instruction. When the hap is triggered, this value is passed as an argument to the hap handlers. This provides the user with a rudimentary way of passing information from the simulated system to the hap handler.

Magic instructions have to be compiled into the binary files that are executed on the target. The file `magic-instruction.h` in `[simics]/src/include/simics/` defines a `MAGIC(`*n*`)` macro that can be used to place magic instructions in your program, where *n* is the immediate value to use.

Note: The declaration of the macros are heavily dependent on the compiler used, so you may get an error message telling you that your compiler is not supported. In that case, you will have to write the inline assembly corresponding to the magic instruction you want to use. The GCC compiler should always be supported.

Note: The magic instruction macro is directly usable only from C and C++; if your program is written in another language, you will have to call a C function that uses the macro, or an assembly function that includes the magic instruction. (If the language supports inline assembly, that can of course be used as well.) For example, in Java it would be necessary to use the JNI interface. Check your compiler and language documentation for details.

A complete list of magic instructions and the range of the parameter *n* is provided in figure 10.1. Note that the parameter is passed through the `eax` register on x86. Earlier Simics versions had different definitions of magic instructions for x86. The old magic instruction can still be used on x86 if the processor is not running in VMP mode.

There are two different encodings of the `rlwimi`-based magic instruction on PowerPC. On 64-bit models, the new encoding is always used; it is also the one generated by the `MAGIC()` and `MAGIC_BREAKPOINT()` macros in `magic-instruction.h` when compiling 64-bit PowerPC code. The old encoding is used on 32-bit models when the *old_rlwimi_magic* attribute is set. When compiling 32-bit PowerPC code, the macros will use the old encoding unless the preprocessor symbol `SIM_NEW_RLWIMI_MAGIC` has been defined.

It is recommended that the new encoding is used with 32-bit PowerPC models and code by setting the appropriate attribute and preprocessor symbol.

Here is a simple example of how to use magic instructions:

```
#include "magic-instruction.h"

int main(int argc, char **argv)
{
        initialize();
        MAGIC(1);                       // tell the simulator to start
```

Target	Magic instruction	Conditions on n	
ARC	`mov 0, n`	$1 \leqslant n < 64$	
ARM	`orr rn, rn, rn`	$0 \leqslant n \leqslant 14$	
ARM Thumb-2	`orr.w rn, rn, rn`	$0 \leqslant n \leqslant 12$	
H8300	`brn n`	$-128 \leqslant n \leqslant 127$	
M680x0	`dbt dx,y`	$0 \leqslant n < 0x3ffff$	
	$x = n[17:15], y = n[14:0] << 1 \mid 1$		
MIPS	`li %zero, n`	$0 \leqslant n < 0x10000$	
PowerPC	`rlwimi x,x,0,y,z`	$0 \leqslant n < 8192$	new encoding
	$x = n[12:8], y = n[7:4], z = n[3:0] \mid 16$		
PowerPC	`rlwimi x,x,0,y,z`	$0 \leqslant n < 32768$	old encoding
	$x = n[14:10], y = n[9:5], z = n[4:0]$		
SH	`mov rn, rn`	$0 \leqslant n < 16$	
SPARC	`sethi n, %g0`	$1 \leqslant n < 0x400000$	
x86	`cpuid`	$0 \leqslant n < 0x10000$	
	with `eax` = $0x4711 + n * 65536$		
Cell SPU	`or n, n, n`	$0 \leqslant n < 128$	

Figure 10.1: Magic instructions for different Simics Targets

```
                                        // the cache simulation
        do_something_important();
        MAGIC(2);                       // tell the simulator to stop
                                        // the cache simulation
        clean_up();
}
```

This code needs to be coupled with a callback registered on the magic instruction hap to handle what happens when the simulator encounters a magic instruction with the arguments 1 or 2 (in this example, to start and stop the cache simulation).

Simics implements a special handling of magic instructions called **magic breakpoints**. A magic breakpoint occurs if magic breakpoints are enabled and if the parameter n of a magic instruction matches a special condition. When a magic breakpoint is triggered, the simulation stops and returns to prompt.

Magic breakpoints can be enabled and disabled with the commands **enable-magic-breakpoint** and **disable-magic-breakpoint** . The condition on n for a magic instruction to be recognized as a magic breakpoint is the following:

```
n == 0 || (n & 0x3f0000) == 0x40000
```

Note that the value 0 is included for architectures where no immediate can be specified. The file `magic-instruction.h` defines a macro called MAGIC_BREAKPOINT that places a magic instruction with a correct parameter value in your program.

10.2 Symbolic Debugging

A vital part of a debugger's task is to understand the system being debugged at a higher level than just machine instructions and memory contents. The user thinks in terms of processes, functions, and named variables, so the debugger presents a view of the software that matches these concepts. This view is even more important in Simics, where the user has access to the whole system and not only user processes. To handle this, Simics uses *context objects*, *symbol tables* and *software trackers*.

10.2.1 Contexts

A **context** object represents a virtual address space. Each processor in the simulated system has a *current context*, which represents the virtual address space currently visible to code running on the processor. Virtual-address breakpoints are properties of contexts; different context objects have separate sets of virtual breakpoints, and by changing a processor's current context, you change its set of virtual breakpoints.

The correctness of the simulation does not depend on contexts in any way; the concept of multiple virtual address spaces is useful for *understanding* the simulated software, but not necessary for just running it. What contexts to create and how to use them is entirely your business; Simics does not care.

By default, every processor in a simulation cell use the same default context. You may create new contexts and switch between them at any time. This allows you, for example, to maintain separate debugging symbols and breakpoints for different processes in your target machine. When a context is used in this manner (active when and only when a certain simulated process is active), the context is said to *follow* the process.

Simics Analyzer will help you coordinate contexts for processes running on the target system.

10.2.2 Symbol Tables

A **symtable** object stores debugging and symbol information for an address space. The information is usually loaded from the same binaries that are used on the target system, and relies on the compiler to create it in the first place.

The symtable object allows Simics to translate back and forth between source code locations and code addresses, variable names and memory locations, and so on. Each context can have a symtable associated with it, which enables symbolic debugging within that context.

Reading Debug Information from Binaries

Symbolic information is normally read from file using the ⟨**symtable**⟩**.load-symbols** command, or directly using the **new-symtable** command. Simics supports reading symbol from

ELF binaries, and the supported debug info formats are DWARF and STABS. Also, the files must be present on the host machine—Simics cannot read directly from the file system of the simulated machine.

When loading the information from a binary, it is possible to adjust the address where the code is loaded. See the documentation of the ⟨**symtable**⟩**.load-symbols** command for more information.

The symtable object uses the debug information to find the source code of the code running in the simulator. This will work automatically if the source can be found in the same place where it was when the binary was compiled. If the directory names do not match, the ⟨**symtable**⟩**.source-path** command can be used to tell Simics how to find the files. The *source path* of a symtable is a semicolon-separated list where each element is either a path name, added to the beginning of source file names, or a string of the form *a*>*b*, replacing *a* with *b* at the beginning of source file names. Both *a* and *b* will be interpreted as directory name.

For example, the source path `/usr/src>/pkg/source;/usr>/misc` would cause the file name `/usr/src/any.c` to be translated to `/pkg/source/any.c`, or, if no such file exists, to `/misc/src/any.c`.

Another example, when moving between different host operating systems is the source path `/usr/src>C:\My\Source` which would translate the file name `/usr/src/any/file.c` to `C:\My\Source\any\file.c`.

The simplest case is when all the sources are in a single directory. To find all the sources, the source path only needs to point to that directory. For example, `/usr/src` would translate the file name `/tmp/file.c` to `/usr/src/file.c`.

Special Considerations

Here are some things to think about when preparing a binary for debugging:

- Some versions of the Sun WorkShop (Forte) C compiler do not put the debug information in the final executable, but expect a debugger to read it from the object files directly. This is not supported by Simics, so be sure to use the `-xs` option when compiling.
- If getting sensible stack traces is important, adhere to the target machine's calling and stack frame conventions. In other words, avoid optimizations such as GCC's `-fomit-frame-pointer`.
- Currently, only C is supported, not C++. It is possible to debug programs built from a mixture of C and C++ source, but then only symbols from the C part (and those declared `extern "C"`) will be reliably recognized, for name mangling reasons.
- It is possible to debug dynamically loaded code by specifying the base address of each module when using **load-symbols**, but it is easier to just link the code statically when possible. See section 14.1 for how to find the base address on some systems.

Loading Symbols from Alternate Sources

Sometimes it is desirable to read symbols from a source other than a binary file—perhaps all you have is a text file listing the symbols. The ⟨**symtable**⟩**.plain-symbols** command reads symbols from a file in the output format of the BSD **nm** command. Example:

```
000000000046b7e0 T iunique
000000000062ba40 B ivector_table
00000000005a6338 D jiffies
```

The hexadecimal number is the symbol value, usually the address. The letter is a type code; for this purpose, D, B, and R are treated as data and anything else as code.

The symbols do not have any C type or line number information associated with them, but you will at least be able to print stack traces and find the location of statically allocated variables and functions.

10.2.3 Finding Relocated Code

If the code is runing at a different memory location compared to that given in the symbol information in the binary, special care needs to be taken when loading symbols and debug information. The ⟨**symtable**⟩**.load-symbols** command takes a start address that tells Simics where the entry point (or text segment start, see the command documentation) is. Finding out where this is can be the hard part, and some common cases are outlined below.

Linux shared objects

A dynamically shared library (usually with the suffix `.so`) is relocated when it is loaded, but it is usually easy to predict where it is loaded. The `ldd` command lists all library dependencies and the expected load addresses.

```
# ldd /bin/ls
        librt.so.1 => /lib/tls/librt.so.1 (0x00128000)
        libacl.so.1 => /lib/libacl.so.1 (0x0013c000)
        ...
```

This means that the address in parentheses after the listed library should be given to the ⟨**symtable**⟩**.load-symbols** command when loading symbols from the library.

If the `ldd` command isn't available, the same results can be achieved by setting the environment variable LD_TRACE_LOADED_OBJECTS to something and running the program. Instead of running it normally, it will print the library dependencies.

```
# LD_TRACE_LOADED_OBJECTS=1 /bin/ls
        librt.so.1 => /lib/tls/librt.so.1 (0x00128000)
        libacl.so.1 => /lib/libacl.so.1 (0x0013c000)
        ...
```

There is one important caveat. Some Linux distributions enable address randomization, which means that the load address will be different every time. If you run ldd several times and get different results, this is probably what is happening. This randomization can be disabled by changing a kernel configuration parameter. Running the following command as root on the target system will disable it:

```
# echo 0 > /proc/sys/kernel/exec-shield-randomize
```

10.2.4 Software trackers

Unlike ordinary debuggers, Simics is a full-system debugger: the debugger is in control of the entire simulated system, and not just one of its processes. However, debugging often involves one or a few specific processes, and it is convenient to hide the rest of the system. Software trackers provides the means to detect when a particular process is running and when it is not running, and to ensure that the correct context is active when the process under inspection is running.

Simics Analyzer provides software trackers for several common operating systems.

10.3 Reverse Execution

Simics has the capability to run a simulation in reverse, which in many cases can greatly simplify the debugging of complicated software problems.

From a user perspective, there is little difference between running the simulation forwards and backwards; all the standard debugging tools like breakpoints and watchpoints can be used when the simulation is run in reverse. There are, however, some reverse execution specific issues, which are discussed in the following sections.

10.3.1 Using Reverse Execution

Before any reverse operations can be performed, there has to be at least one time bookmark; reverse operations are possible in the region following the first (oldest) bookmark. Depending upon how Simics is invoked (and various user preferences), a time bookmark denoted "start" is sometimes added automatically at the beginning of the simulation. Bookmarks can also be created by enabling reverse execution in an external debugger.

Bookmarks are managed through the commands

- **set-bookmark** *label*,
- **list-bookmarks** and
- **delete-bookmark [***label* | **-all]**.

Note: From a performance and resource utilization perspective, it might be advantageous to put the first bookmark as close to the region of interest as possible.

Reverse execution support is disabled when all time bookmarks are deleted.

Once reverse execution has been enabled (by the creation of a time bookmark), it is possible to both run the simulation in reverse and to skip backwards in time. The Simics commands to do this are

- **reverse**,

- **reverse-to** *position* and
- **skip-to** *position*.

The **reverse** and **reverse-to** commands run the simulation backwards until a breakpoint occurs or till the oldest time bookmark is reached. The **skip-to** command jumps to a particular point in time, ignoring intermediate breakpoints. It should be noted that skipping can be significantly faster than reversing.

The **skip-to** and **reverse-to** commands take either an absolute step count or a time bookmark as argument.

Most forward executing commands used for debugging has a corresponding reverse variant obtained by adding a reverse prefix. The reverse variant of **step-instruction** is for instance **reverse-step-instruction**.

Any external input (like a human typing on a virtual serial console) is replayed when the simulation is being run forward after a reversal; this guarantees that the simulation will follow the same path as it did originally. For the same reason, all external input is ignored until the point is reached where the first reverse operation was initiated. It is possible to override this behavior with the **clear-recorder** command. This command discards all recorded input and allows an alternate future to take place.

Some changes to the simulation from entities outside the simulation are not recorded by recorders, for example manual changes from the command line interfaces or the graphical user interface. Such changes can make reverse execution operate somewhat unpredictably. It is recommended that any time bookmarks before a non-replayable change of the simulation state is deleted explicitly.

10.3.2 Performance

The usage of time bookmarks has a certain impact on overall performance since it implicitly enables reverse execution support. Normal performance is always obtained if all time bookmarks are removed.

The reverse execution engine optimizes performance for certain reverse operations that are expected to be common. One example of this is that skipping to a bookmark (or to the region just after a bookmark) can be significantly faster than skipping to some other location.

It is possible to tune certain reverse execution parameters in order to optimize for a particular usage pattern (although the default settings should work well in most situations). Tradeoffs exists between:

- reverse performance
- forward performance
- memory utilization
- scope of reversibility

The **rexec-limit** command is the primary tool for adjusting the balance, e.g.

```
simics> rexec-limit steps = 20000000
```

```
simics> rexec-limit size_mb = 200
```

The steps limit indicate that the scope of interest is at most the specified number of steps. By imposing a steps limit, resources can be spent more effectively with the drawback that reversal past the limit may not be possible.

The size limit imposes a limit on the amount of memory reverse execution may use. If the limit is exceeded, reverse performance will be traded for less memory consumption.

Chapter 11

Using Simics for Hardware Bring-Up and Firmware Development

Simics makes hardware bring-up, firmware development, and other low-level programming tasks easier in a number of ways:

Hardware replacement
A simulator replaces hardware. This has two key benefits during hardware bring-up: you can start working on the software before the hardware is available, and you can have as many copies of the simulated hardware as you like. Both of these translate directly to reduced total development time for the combined hardware+software product.

Inspection and modification
You can inspect the state of the entire simulation—memory, processor registers, device registers, anything—all entirely non-intrusively. And time is simply paused while you do so. You can run backward in time, modify memory or register contents, and then run forward again and see the effects of this change.

Full debug support
The full power of Simics debugging (see chapter 10), with breakpoints, symbolic debugging, reverse execution, scripting, and so on, is available everywhere, even at the very lowest levels.

11.1 A Simple Example

It is easy to write a handful of instructions directly to memory, fill the registers with any necessary values, and manually single-step through this little program:

```
simics> load-file test.bin 0xfff10000
simics> set-pc 0xfff10000
simics> %r17 = 4711
simics> si
[mpc8641d_simple.soc.cpu[0]] v:0xfff10004 p:0x0fff10004  xor r1,r1,r1
```

```
simics> si
[mpc8641d_simple.soc.cpu[0]] v:0xfff10008 p:0x0fff10008  xor r2,r2,r2
```

As always in Simics, this kind of thing can be scripted if you expect to run it more than once:

test.simics:

```
run-command-file targets/mpc8641-simple/firststeps.simics
load-file test.bin 0xfff10000
set-pc 0xfff10000
%r17 = 4711
continue 12
expect %r17 4713
expect %pc 0x1001c
quit
```

```
$ ./simics test.simics
** Values differ in expect command: 4711 4713
$
```

Here, we first call another simics script to set up the machine for us, then run our little test case. The **expect**s will cause Simics to exit with an error code (as shown) if the conditions are not met; otherwise, the **quit** will cause Simics to quit successfully.

11.2 Going Further

The simple script in the last section can be extended in several directions:

- **load-file** simply writes the contents of a file directly to memory. There are at least two other options:
  - Using **set** to write values directly to memory. This is useful if the test program is truly just a few instructions long.
  - Using **load-binary** to load an executable in one of the formats Simics recognizes, such as ELF. Unlike **load-file**, this command automatically loads the executable at the right address, and returns the entry point address.
- You can have more complicated stop conditions than simply "run twelve instructions"; for example, you can use execution or data breakpoints (section 10.1.1), control register breakpoints (section 10.1.3), device I/O breakpoints (section 10.1.4), or magic instruction breakpoints (section 10.1.7).
- Various conditions cause Simics to trigger *haps*; for example breakpoints, privilege level changes, magic instructions, and traps. You can easily write a small hap callback

function that gets called whenever this happens; such a callback could terminate the simulation (indicating success or failure), or simply log or change some value.

Chapter 12

Using Simics with Wind River Tornado or Workbench

Simics can function as a debugger backend to Wind River's Tornado and Workbench IDEs.

The normal way to control a target machine from these IDEs is to have a *WDB agent* running on the target. It talks to the IDE using the Wind River Debug (WDB) protocol, and carries out low-level debugging tasks such as starting, stopping, setting breakpoints, and reading memory.

Simics comes with a built-in WDB agent, **wdb-remote**. Just like a WDB agent that runs on the target machine, it listens for commands on a UDP port, and can start, stop, step, and inspect the target. However, unlike an agent running on the target, it is completely non-intrusive since the whole protocol is handled by the simulator. No debugging code runs on the target, and it is unaware of being halted since it is the entire simulation that stops. You can single-step, use breakpoints and inspect memory contents, and the target will never know.

12.1 Starting the Simics WDB Agent

Instances of **wdb-remote** are created with the **new-wdb-remote** command. At the simics console, do for example

```
simics> new-wdb-remote processor = cpu0 cpu-type = PPC603 \
        mem-size = 0x1000000 host-pool-base = 0x0 \
        host-pool-size = 0x10000
```

Use the tab completion to get suggestions for possible *processor* and *cpu-type* arguments. *mem-size* is the number of bytes the agent will claim that the target has; it is not vital to get this right. *host-pool-base* and *host-pool-size* define a client-owned area of the target's memory; if you are not going to download code onto the target or something like that, this area will not be used, and you can specify an arbitrary region, for example the one given here.

Some WDB clients, most notably some Wind River Workbench and Tornado clients, require some of the optional parameters of **new-wdb-remote** to be specified. If not specified correctly, these clients will not recognize the simics WDB agent as supported.

The optional parameters in question are *rt-name*, *rt-version* and *rt-type*. When debugging a target machine running VxWorks 6.8 from Wind River Workbench these parameters should be set to "VxWorks", "VxWorks 6.8" and "WDB_RT_VXWORKS" respectively. The complete command invocation then becomes:

```
simics> new-wdb-remote processor = cpu0 cpu-type = PPC603 \
        mem-size = 0x1000000 host-pool-base = 0x0 \
        host-pool-size = 0x10000 rt-name = "VxWorks" \
        rt-version = "VxWorks 6.8" rt-type ="WDB_RT_VXWORKS"
```

Note that these values are not used by the Simics WDB agent, but is just passed to the WDB client when requested, they do not affect the functionality of the Simics WDB agent. Also note that as these values are used to satisfy the WDB client, they should match the VxWorks/Workbench version you are using. Simics will not verify that these values correspond to the target machine's software.

For a more thorough explanation, including optional parameters, see the documentation for the **new-wdb-remote** command in the *Reference Manual*.

12.2 Connecting Tornado

First create a **wdb-remote** agent, as described in section 12.1. Then let Tornado connect to it:

1. From the Tornado menu, choose **Tools** → **Target Server** → **Configure**.
2. Press *New* to create a new target server configuration.
3. Choose **Target Server Properties** → **Target Server File System**; make sure that *Enable File System* is not enabled.
4. Choose **Target Server Properties** → **Back End**; select *wdbrpc*.
5. Choose **Target Server Properties** → **Core File and Symbols**; select *File*, and fill in the name of your VxWorks image.
6. For *Target Name/IP Address*, fill in the name of the host where Simics is running, or `localhost` if it is the same computer that you run Tornado on.
7. Press *Launch*.

Thankfully, most of these steps only have to be carried out the first time you connect.

Tornado is now connected to Simics. For example, you can inspect the target with the browser (**Tools** → **Browser**). (Note that if you have not let the target machine boot before you try this, or if the target machine does not run VxWorks, parts of the browser will not work.)

12.3 Connecting Workbench

This section describes how to connect Simics to Wind River Workbench 3.3. The instructions apply to earlier Workbench versions as well, but some details may differ.

Workbench is based on Eclipse, so you may first want to install the Simics Eclipse plug-ins to easily launch Simics from within Eclipse. For instructions on how to install the Simics plug-ins, see the chapter *Installing the Simics Eclipse Tools* in the *Installation Guide*.

Once you have launched Simics, create a **wdb-remote** agent, as described in section 12.1. Since the agent runs outside the simulated system, you can start and connect to the agent before booting the system.

Now let Workbench connect to it:

1. Create a new connection: Open the **Device Debug** perspective and select **New Connection...** from the Target menu. Choose *Wind River VxWorks 6.x Target Server Connection*.

2. In the dialog that appears, fill in the hostname of the computer you are running Simics on, or `localhost` if it is the same computer that you run Workbench on. Make sure the *Bypass checksum comparison* option is enabled. As *Kernel image*, select the OS image loaded by your Simics script. Press **Next** twice.

3. Make sure that *Query target object lists* and *target object states on connect* is not enabled. Press **Finish**.

4. A new item for your connection now appears in the *Remote Systems* view. The item contains two sub-items, **Wind River Target Debugger** and **Wind River Launches**. The connection autoconnects when you first create it. If you want to reconnect later, you can either right-click the icon and choose **Connect**, or double-click a launch that appears under **Wind River Launches**.

5. Once the connection is established, the **Wind River Target Debugger** item has a sub-item, named like *VxWorks 6.x:CPU [stopped]*. Right-click this item and select **Debug rarr; Attach to System VxWorks 6.x (System Mode)**.

Workbench is now connected to Simics; you should see a disassembly of the memory region surrounding the current address of the program counter. You can now run, stop or single-step the simulation.

12.4 Limitations

Unlike an agent running inside the target OS, **wdb-remote** has no OS awareness. This means that it cannot operate on specific VxWorks or Linux tasks; it only supports the system context. For example, it is only possible to set breakpoints, single step, and suspend the whole target machine, not individual tasks.

Chapter 13

Using Simics with CodeWarrior

This chapter provides instructions on how to use Simics from the CodeWarrior IDE when debugging a p4080 model.

13.1 About the Simics CodeWarrior Add-on Module

The Simics CodeWarrior Add-on module is a shared library implementing the CodeWarrior CCSSIM2 interface using Wind River Simics as the simulator back-end. It is compatible with Simics models using the e500 and e500-mc cores. However at this point, it is primarily intended to be used with the p4080 model. It should be used with CodeWarrior PA version 10.0.

The Simics CodeWarrior module is available for Windows and Linux hosts. Note that on Linux, Simics is available as either a 32-bit or a 64-bit application. You can use either version of Simics with CodeWarrior, but since the CodeWarrior IDE itself is only available as a 32-bit application, you should use the 32-bit version of the Simics CodeWarrior Add-on module no matter which version of Simics you are using.

13.2 Install Instructions

1. In order to use Simics and the P4080 model with CodeWarrior, please make sure that you have installed the following packages.

Package ID	Package Name	Windows	Linux (32-bit)	Linux (64-bit)
1000	Simics-Base	win32	linux32	linux64
2067	P4080	win32	linux32	linux64
4012	P4080-Images	win32	linux32	linux64

   Below, it is assumed that Simics is installed in: `[simics-4.2.x]`, the P4080 model in `[simics-p4080]`, and CodeWarrior PA 10.0 in `[codewarrior]`.

2. To verify that Simics and the P4080 model are installed correctly, please launch Simics with the below script, and check that no errors are reported by Simics. `[simics-p4080]\targets\p4080-simple\p4080-simple-linux-common.simics`

13.3 Setup Instructions

With everything installed, we will now proceed to setting up CodeWarrior for debugging the P4080 model in Simics.

1. Launch the CodeWarrior PA 10.0 IDE.

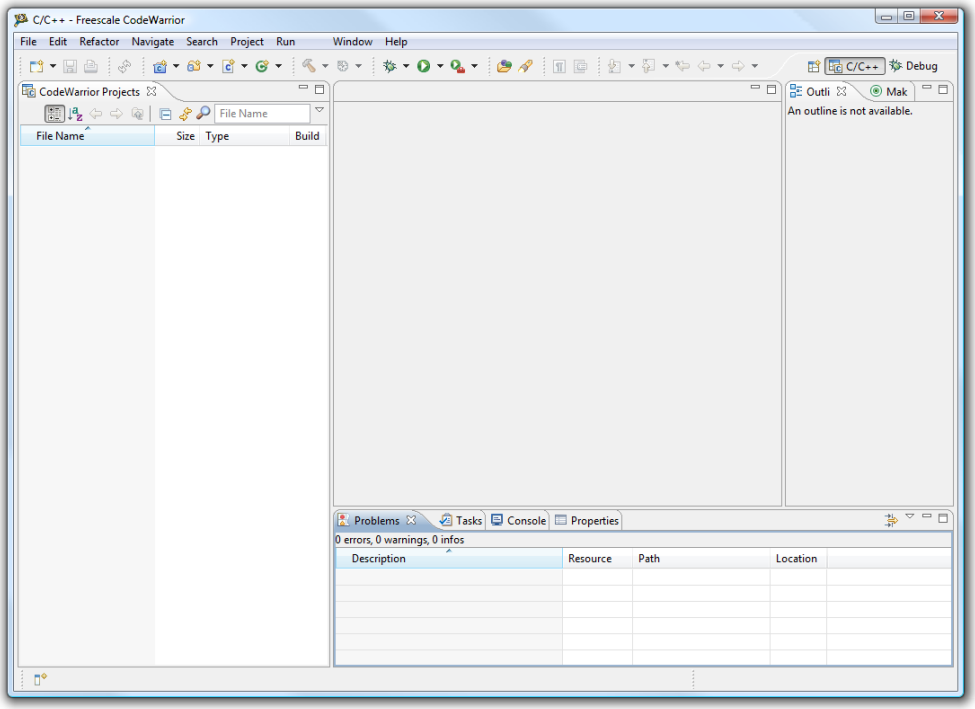

Figure 13.1: The CodeWarrior PA 10.0 IDE Main Window

2. Create a new project in CodeWarrior:

   - Select **File** → **New** → **Power Architecture Project**.
     In the resulting dialog, enter a name for the project and click next.
   - On the next page, select the **P4080** processor and use the **Bareboard Application** toolchain. Then click next.
   - Enter C as the language, click next.
   - Select the **Simulator** debugger connection type and click next.
   - Select **Simics P4080** as the simulator, and click finish.
   - The project will now be created and should be visible in the CodeWarrior projects view.

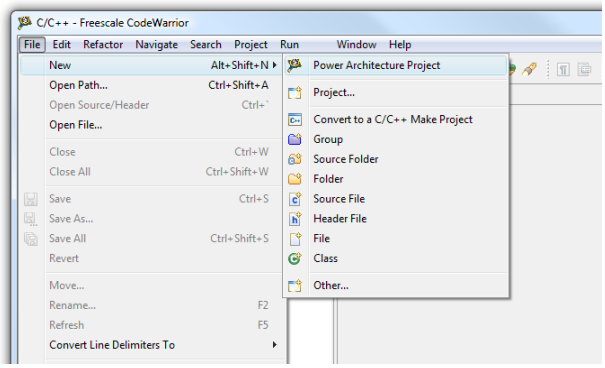

Figure 13.2: New Project

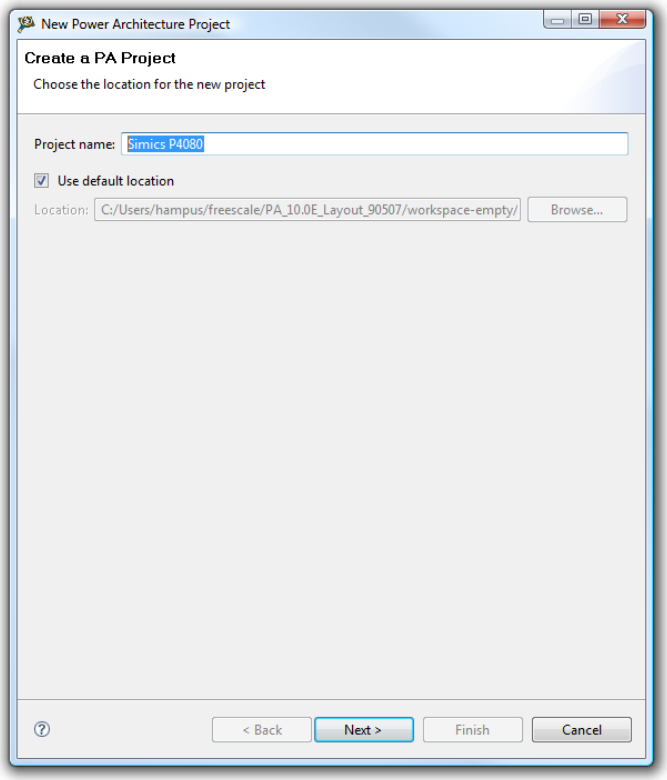

Figure 13.3: New Project: Name and Location

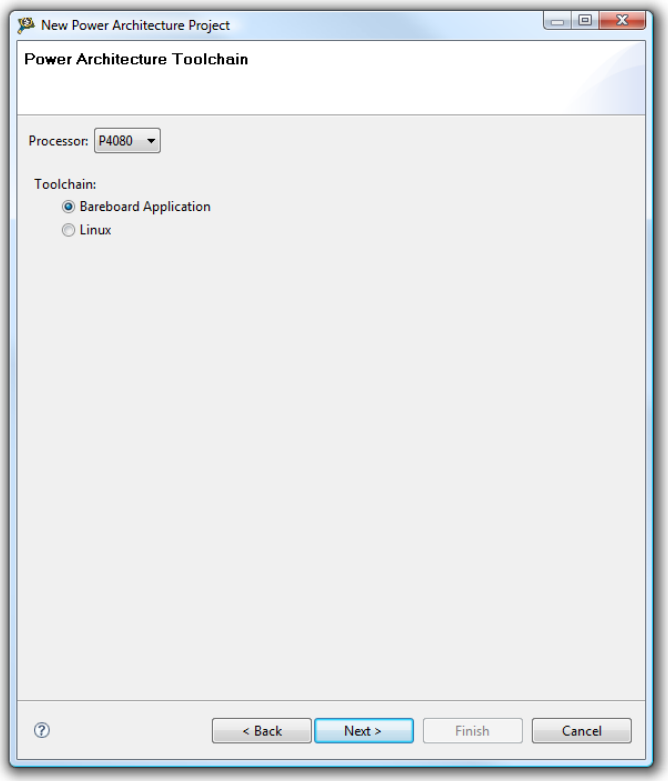

Figure 13.4: New Project: Processor and Toolchain

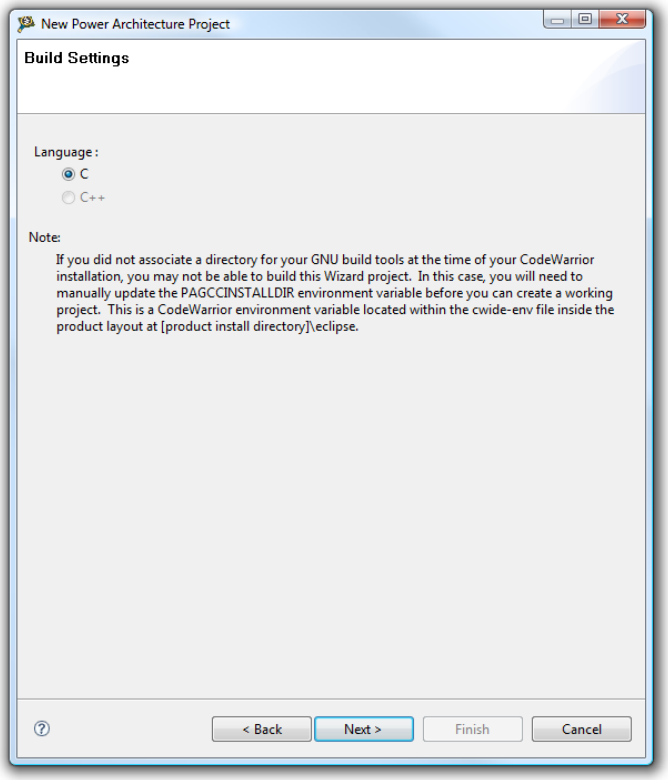

Figure 13.5: New Project: Implementation Language

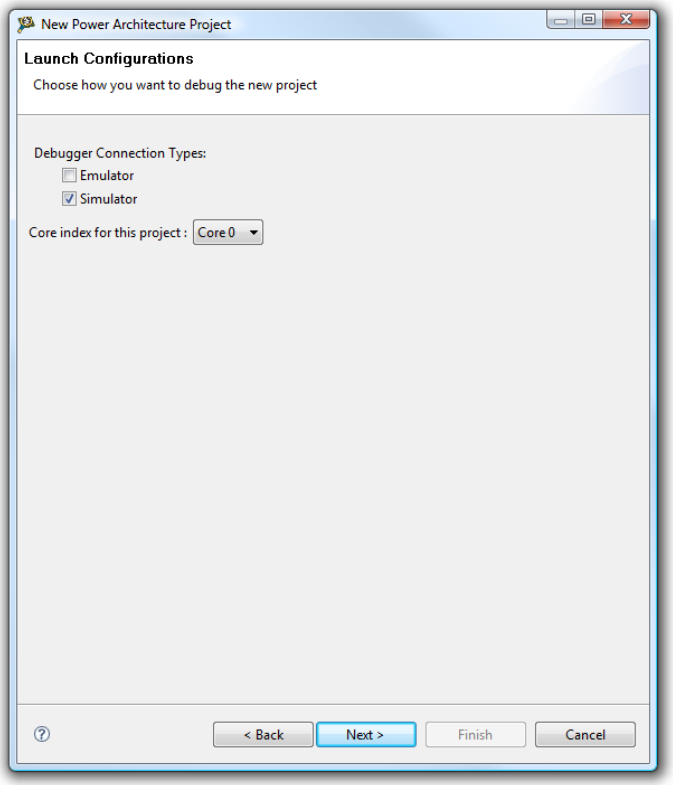

Figure 13.6: New Project: Launch Configurations

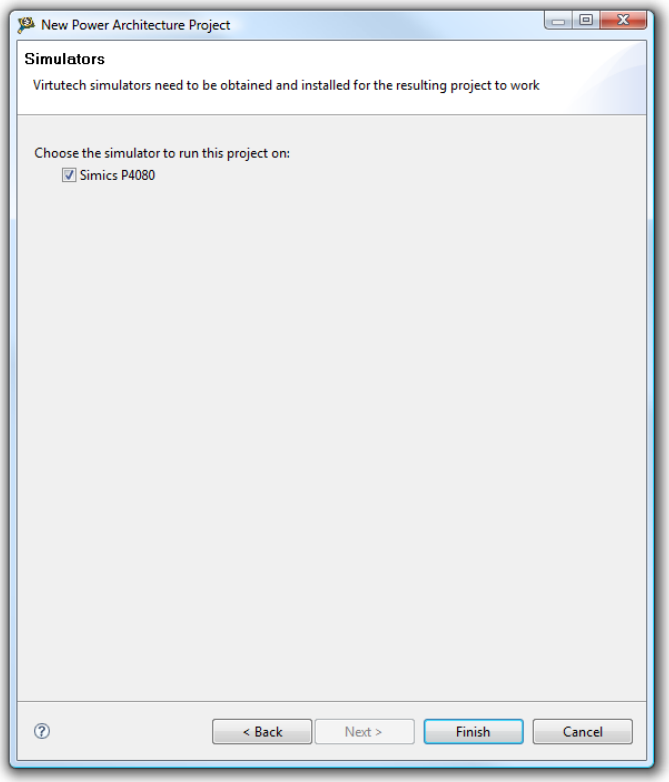

Figure 13.7: New Project: Simulators

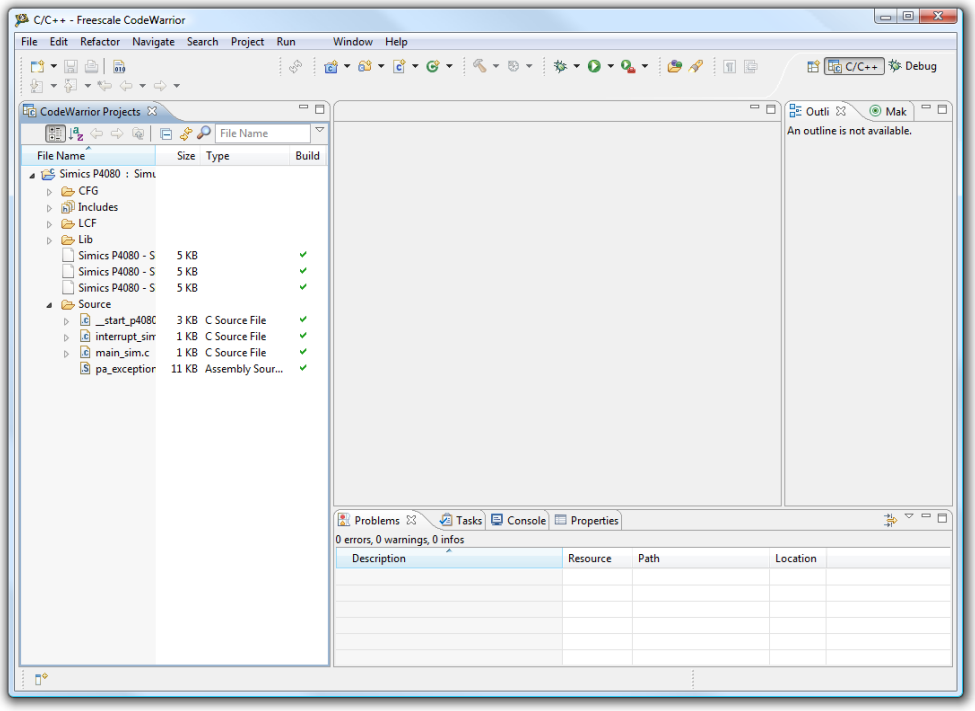

Figure 13.8: CodeWarrior Main Window with New Simics P4080 Project

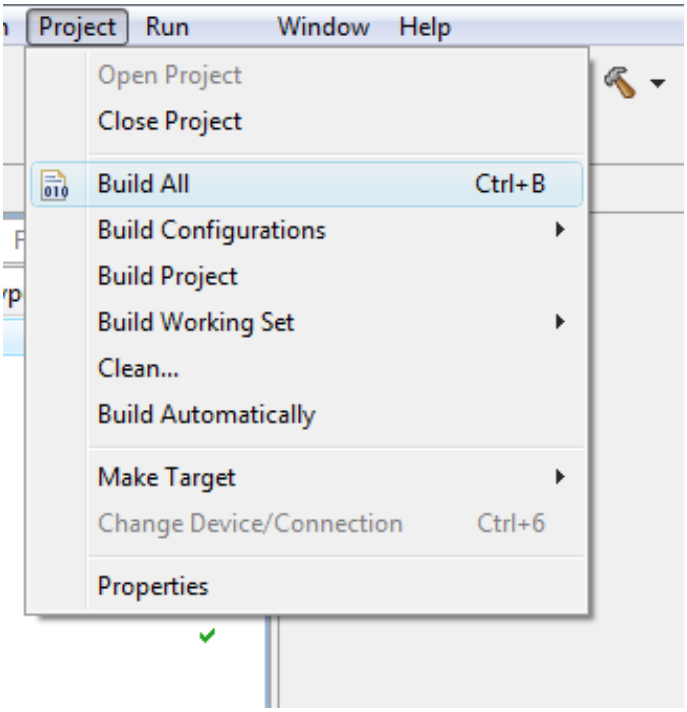

Figure 13.9: Build Project

3. The newly created project contain a sample application the we will later download to the P4080 model and debug, but first we must build it. To build the application in the project, select **Project** → **Build All**.

   A dialog with a progress bar is presented. Once the build process has completed the build application will appear in the project.

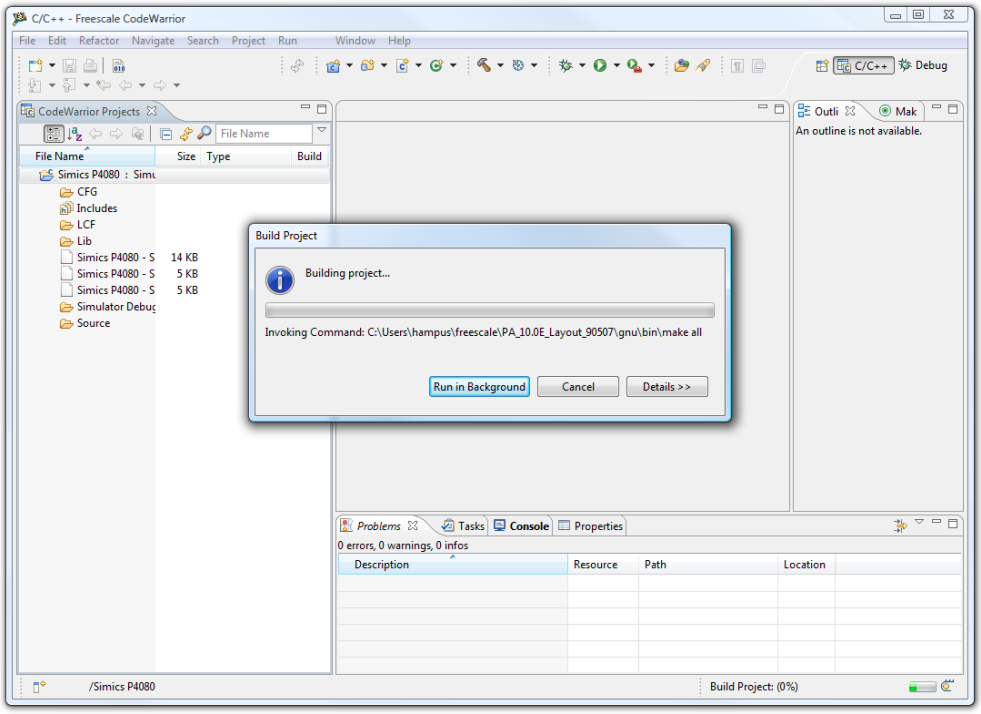

Figure 13.10: Build Project: Progress Bar

4. Next we will modify the launch configuration to find Simics and the P4080 model.

   - Select **Run** → **Open Debug Dialog...**. On the left hand side in the dialog, expand the section **CodeWarrior Download** and select the row **Simics P4080 - Simulator Debug - Simics P4080**. Then on the right hand side, select the **Debugger** tab, and inside it, the **Simics Settings** tab. Here we must specify the paths to the Simics executable, the model script and the CodeWarrior add-on library. Follow the instructions in the dialog to select the correct files as installed on your computer.

     Note: When specifying the path to the CodeWarrior add-on, please specify the following path (regardless of the example in the dialog), replacing [simics-4.2.x] with the installation path for the Simics Base package:

     On Linux: `[simics-4.6.x]/linux32/bin/libp4080iss.so`

     On Windows: `[simics-4.6.x]/win32/bin/libp4080iss.dll`

   - Finally, click the **Debug** button. This will close the dialog window and launch the configuration.

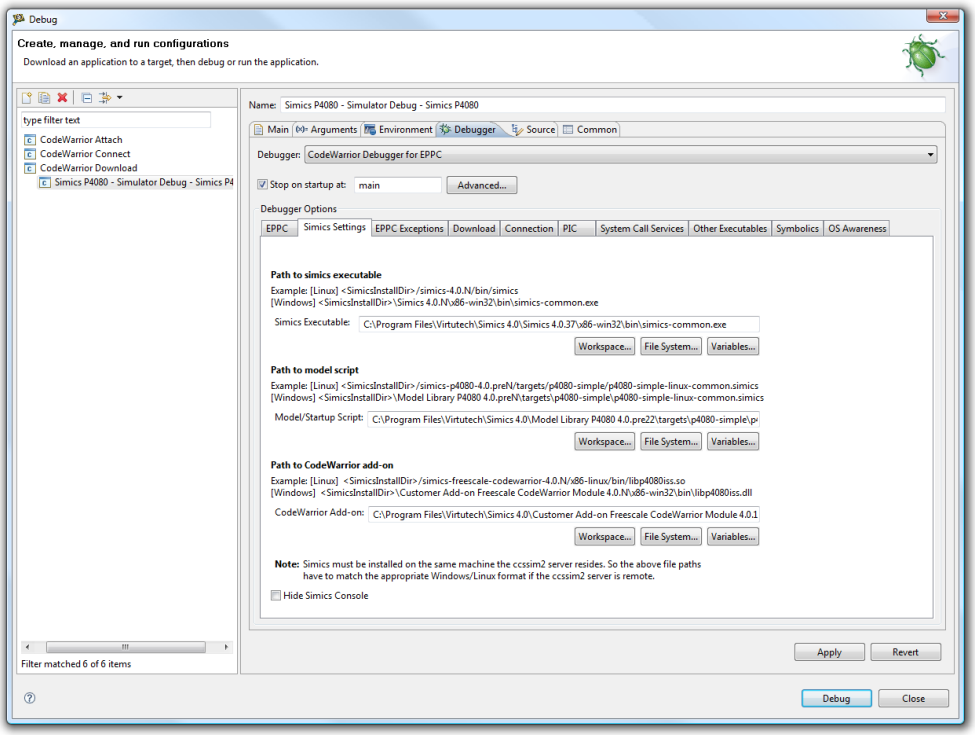

Figure 13.11: Debug Dialog: Specify Location of Simics

13.4 Usage Instructions

Simics will now launch in the background, and the debugger will be connected to the P4080 model.

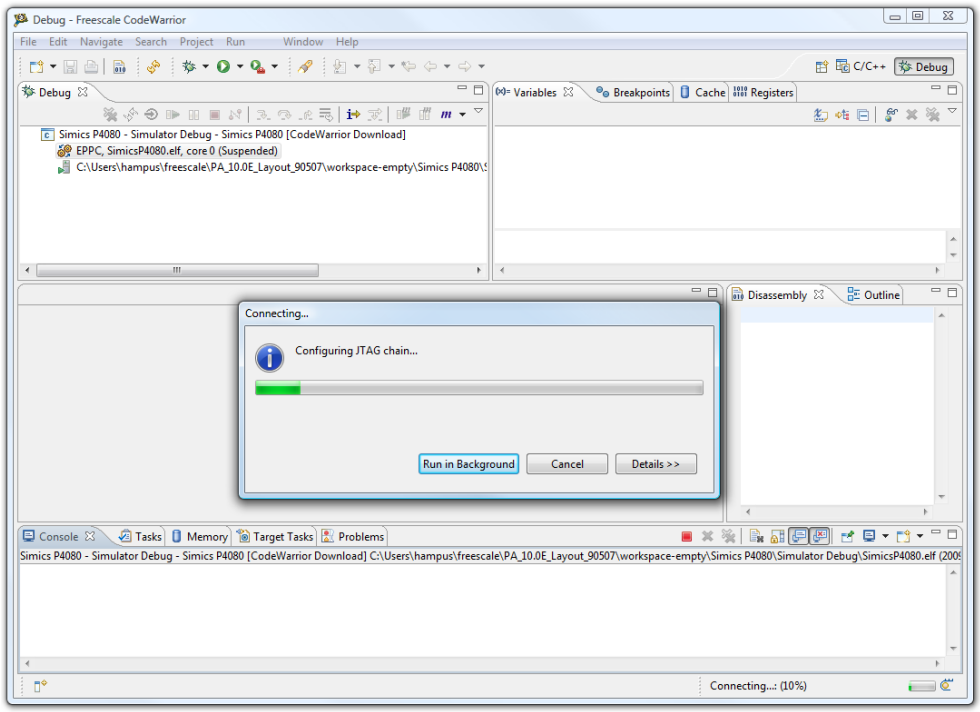

Figure 13.12: Connecting to Simics

The application will also be downloaded to the memory of the P4080 model.

Once the download has completed, the debugger will be positioned to the starting point of the application. From here, the application can be run or debugged just as the real P4080 hardware board. E.g. you can step through the source code, inspect variables and registers, set breakpoints and run until they trigger.

If your application writes to the targets serial consoles, these printouts will appear in the target console windows. If you run an application that requires user input, you can also interact with the simulated target via the target consoles.

The debug session can be terminated via the **Terminate** tool-bar button or menu item. This will also shut down the Simics process.

The same launch configuration can later be relaunched without further configuration, by selecting **Run** → **Debug History** → **Simics P4080 - Simulator Debug - Simics P4080**.

13.5 Extra Logging

Trace logging in the Simics implementation the ccssim2 api can be enabled by setting the following environment variables to the name of a log file: `export SIMICS_CW_`

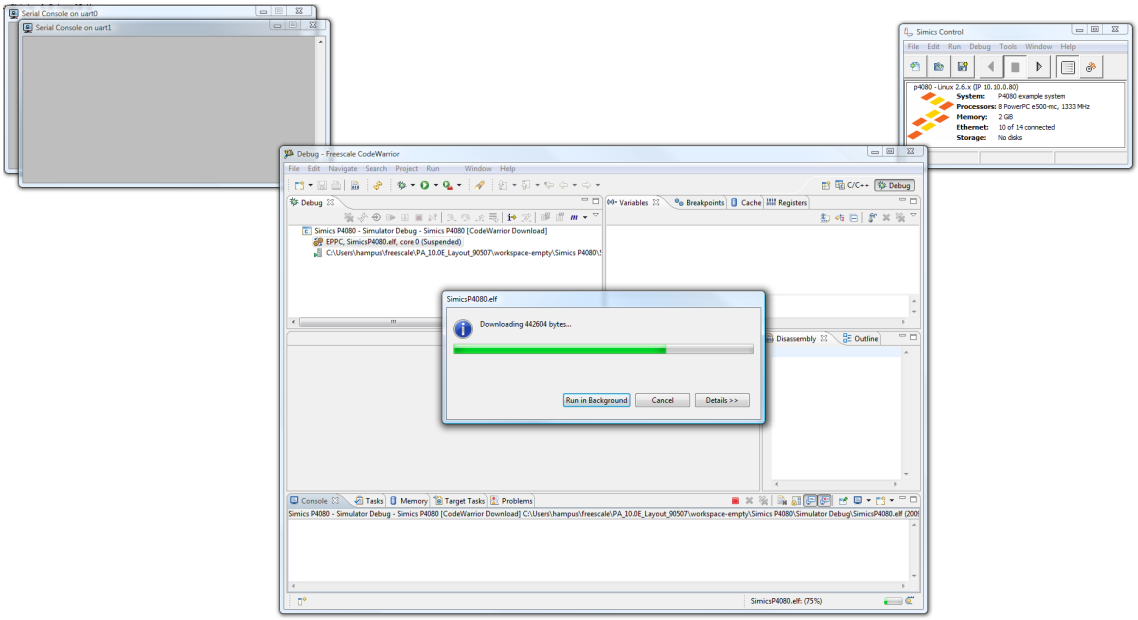

Figure 13.13: Downloading Application

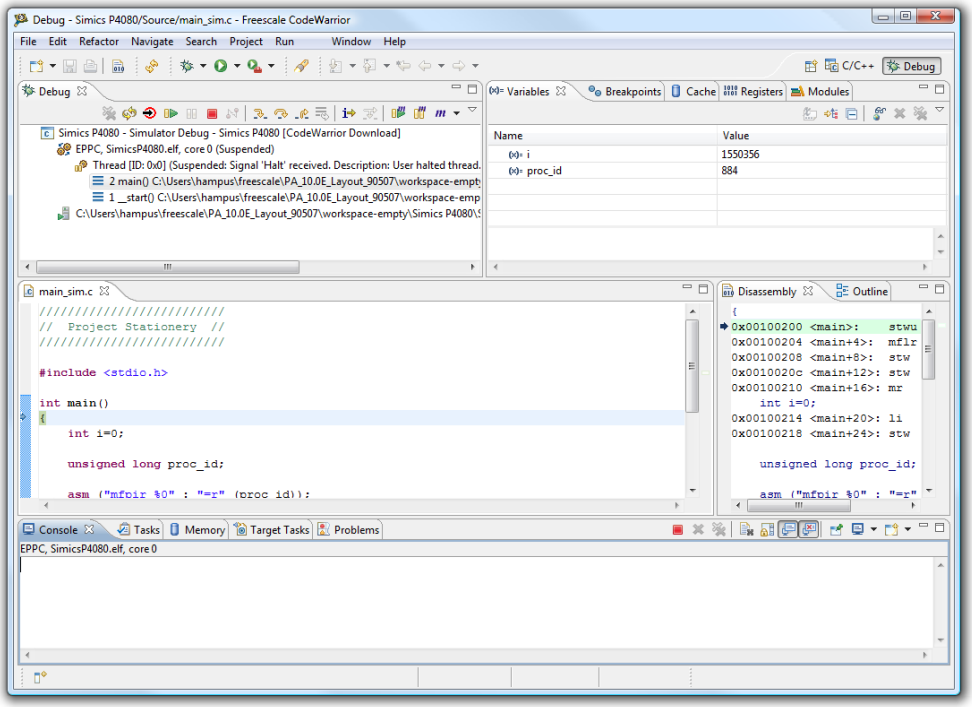

Figure 13.14: Debugger Ready

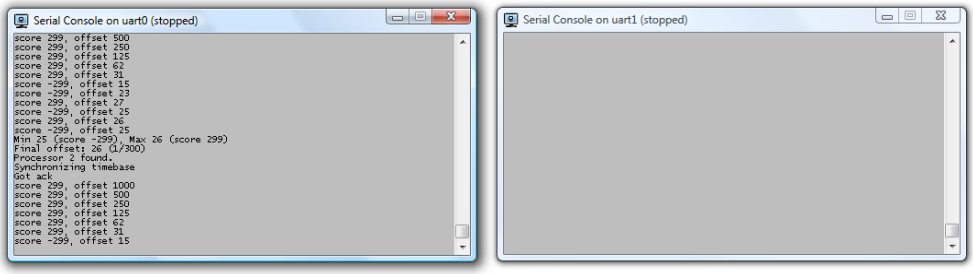

Figure 13.15: Simics Target Consoles

`LOGFILE=~/tmp/cw-logfile.txt` To get a more detailed log, useful when debugging our CodeWarrior module change the environment variable SIMICS_CW_LOGLEVEL to DEBUG: `export SIMICS_CW_LOGLEVEL=DEBUG` Logging of protocol messages from the Simics Frontend Protocol can be enabled by setting the following environment variables to the name of a log file: `export SIMICS_SFP_LOGFILE~/tmp/sfp-logfile.txt`

Chapter 14

Using Simics with GDB

This chapter describes how to use **gdb-remote**, a Simics module that lets you connect a GDB session running on your host machine to the simulated machine using GDB's remote debugging protocol, and use GDB to debug software running on the target machine.

If you load the **gdb-remote** module in Simics, you can use the remote debugging feature of GDB, the GNU debugger, to connect one or more GDB processes to Simics over TCP/IP. In order to do this, you need a GDB compiled to support the simulation's target architecture on whichever host you're running. The **gdb-remote** module only supports version 5.0 or later of GDB, but other versions may work as well. Unfortunately GDB's remote protocol does not include any version checking, so the behavior is undefined if you use other versions. For information on how to obtain and compile GDB, see section 14.3.

Note: In order to use GDB with Simics running on a Windows host you need to either compile GDB for Cygwin, or run GDB remotely from a UNIX/Linux machine. The examples in this chapter assume the former.

To connect a GDB session to Simics, start your Simics session and run the **new-gdb-remote** command, optionally followed by a TCP/IP port number, which defaults to 9123 otherwise. This will automatically load the **gdb-remote** module.

When a configuration is loaded, Simics will start listening to incoming TCP/IP connections on the specified port. Run the simulated machine up to the point where you want to connect GDB. To inspect a user process or dynamically loaded parts of the kernel, the easiest solution might be to insert magic instructions at carefully chosen points. For static kernel debugging, a simple breakpoint on a suitable address will solve the problem.

Note: When debugging the start-up phase of an operating system, it might happen that gdb gets confused by the machine state and disconnects when you try to connect. In this case, execute a few instructions and try again.

Once Simics is in the desired state, start your GDB session, load any debugging information into it, and then connect it to Simics using the **target remote** ***host:port*** command, where *host* is the host Simics is running on, and *port* is the TCP/IP port number as described above. Here is a short sample session using *firststeps*:

```
(gdb) symbol-file targets/mpc8641-simple/images/vmlinux-2.6.34-vt-2010-08-25
Reading symbols from /home/conradi/simics-4.4/targets/mpc8641-simple/images/vmlinux-2.6
(gdb) target remote localhost:9123
Remote debugging using localhost:9123
ppc6xx_idle () at arch/powerpc/kernel/idle_6xx.S:147
147             isync
Current language:  auto; currently asm
(gdb)
```

Note: For some architectures, you need to give a command to GDB before connecting (the **set architecture** command in the session above). These are tabulated in the reference manual's section on **gdb-remote**, and will also be printed on the Simics console when you run **new-gdb-remote**.

From this point, you can use GDB to control the target machine by entering normal GDB commands such as **continue**, **step**, **stepi**, **info regs**, **breakpoint**, etc.

Note that while a remote GDB session is connected to Simics, the Simics prompt behaves a little differently when it comes to stopping and resuming the simulation. While the GDB session is at prompt, it is impossible to continue the simulation from within Simics (e.g., by using the **continue** command). However, once you continue the execution from GDB, you can stop it from GDB (by pressing control-C), which causes the simulation to stop and makes both GDB and Simics return to their prompts, or you can stop the simulation from the Simics prompt (also by pressing control-C if Simics was run from the CLI or typing stop if running in a GUI). This only makes Simics return to prompt, while GDB will still think the target program is running. In this state, you should continue the simulation from the Simics prompt before attempting to use GDB.

You can also force GDB back to prompt using the **gdb0.signal 2** command in Simics, which tells the GDB session that the simulated machine got a `SIGINT` signal. **gdb0** here refers to the configuration object created on the fly when the GDB session connected to Simics. You can connect several GDB sessions to one Simics; each connection will be associated to one **gdb*nn*** object.

Since GDB is not the most stable software, especially when using remote debugging, it unfortunately hangs now and then. To force Simics to disconnect a dead connection, you can use the **gdb0.disconnect** command.

Note that the **gdb-remote** module does not have any high-level information about the OS being run inside Simics. This means that in order to examine memory or disassemble code, the data or code you want to look at has to be in the active TLB.

Note: When using **gdb-remote** with targets supporting multiple address sizes (such as x86-64 and SPARC), you must have a GDB compiled for the larger address size. For SPARC, run GDB's configure script with the `--target=sparc64-sun-solaris2.8` option.

14.1 Remote GDB and Shared Libraries

It takes some work to figure out how to load symbol tables at the correct offsets for relocatable object modules in GDB. This is done automatically for normal (non-remote) targets, but for the remote target, you have to do it yourself. You need to find out the actual address at which the shared module is mapped in the current context on the simulated machine, and then calculate the offset to use for GDB's **add-symbol-file** command.

To find the addresses of the shared libraries mapped into a process' memory space under Solaris, use the **/usr/proc/bin/pmap *pid*** command. The start address of the text segment can be obtained from the `Addr` field in the `.text` line of the output from **dump -h *file***.

Under Linux, the list of memory mappings can be found in the file `/proc/`*pid*`/maps` (plain text format). The `VMA` column of the `.text` line of the output from **objdump -h *file*** contains the start address of the text segment.

Using these two values, *map address* and *text address*, you should use *map address* + *text address* as the offset to **add-symbol-file** (it has to be done this way to compensate for how GDB handles symbol loading).

To show you how it works, we will work through a simple example. The example uses a simple program using a really simple shared library. The program can be found in *[firststeps]*`/targets/mpc8641-simple/images/hello` and the shared library is the `libgreeter.so` file in the same directory. Here and in the rest of this section *[firststeps]* refers to the location where the Firststeps package is installed.

Start by booting the firststeps machine. Then mount the host file system and copy the program and shared library onto the machine. This should be done on the simulated machines prompt:

```
~ # mount /host
~ # cp /host/[firststeps]/targets/mpc8641-simple/images/hello .
~ # cp /host/[firststeps]/targets/mpc8641-simple/images/libgreeter.so .
```

Then run the program in the background. The program will enter the infinite loop in the shared library.

Now we need the *map address* and the *text address* of the shared library. To get the map address, get the process's process id with ps and look in the process file system to see where it has mapped the shared library:

```
~ # ./hello &
~ # ps
  PID TTY          TIME CMD
    :
  988 root      1792 R    ./hello
~ # cat /proc/988/maps
00100000-00103000 r-xp 00000000 00:00 0          [vdso]
0fe68000-0ffb7000 r-xp 00000000 01:00 6197       /lib/libc-2.8.so
0ffb7000-0ffc7000 ---p 0014f000 01:00 6197       /lib/libc-2.8.so
0ffc7000-0ffcb000 r--p 0014f000 01:00 6197       /lib/libc-2.8.so
0ffcb000-0ffcc000 rw-p 00153000 01:00 6197       /lib/libc-2.8.so
```

```
0ffcc000-0ffcf000 rw-p 00000000 00:00 0
0ffdf000-0ffe0000 r-xp 00000000 01:00 4114       /root/libgreeter.so
0ffe0000-0ffef000 ---p 00001000 01:00 4114       /root/libgreeter.so
0ffef000-0fff0000 rw-p 00000000 01:00 4114       /root/libgreeter.so
10000000-10001000 r-xp 00000000 01:00 4115       /root/hello
10010000-10011000 rw-p 00000000 01:00 4115       /root/hello
48000000-4801e000 r-xp 00000000 01:00 6194       /lib/ld-2.8.so
4801e000-48020000 rw-p 00000000 00:00 0
4802e000-4802f000 r--p 0001e000 01:00 6194       /lib/ld-2.8.so
4802f000-48030000 rw-p 0001f000 01:00 6194       /lib/ld-2.8.so
bfce9000-bfd0a000 rw-p 00000000 00:00 0          [stack]
```

From this output you can see that the program is running with PID 988 and that the map address is 0xffdf000. The exact PID may differ, adapt the commands accordingly.

To get the *text address* we use objdump. This should be run on a host computer with objdump installed:

```
> objdump -h [firststeps]/targets/mpc8641-simple/images/greeter.so
images/greeter.so:     file format elf32-big

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
    :
  8 .text         000002c0  00000390  00000390  00000390  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
    :
```

The `.text` symbols starts at address `0x390` and this is what we call the *text address*, so if we connect GDB to Simics we have to add the symbols with an offset of `0xffdf000 + 0x390 = 0xffdf390`. Normally you cannot rely on a program on a multicore system such as firststeps to always run on the same processor. However the `hello` example program sets the CPU affinity to force Linux to schedule it on the first processor, **mpc8641d_simple.soc.cpu[0]**.

Create an gdb-remote object in Simics for the first processor:

```
simics> new-gdb-remote cpu = mpc8641d_simple.soc.cpu[0]
```

Now we can set up GDB, connect it to Simics, and debug the program:

```
(gdb) dir [firststeps]/targets/mpc8641-simple/
Source directories searched: [firststeps]/targets/mpc8641-simple:$cdir:$cwd
(gdb) add-symbol-file [firststeps]/targets/mpc8641-simple/images/libgreeter.so 0xffdf390
add symbol table from file "[firststeps]/targets/mpc8641-simple/images/libgreeter.so" at
        .text_addr = 0xffdf390
(y or n) y
```

```
Reading symbols from [firststeps]/targets/mpc8641-simple/images/libgreeter.so...done.
(gdb) target remote localhost:9123
Remote debugging using localhost:9123
14              while (loop); /* Loop until the loop variable is reset by gdb */
Current language:  auto; currently c
(gdb)
```

This is just a toy program written to make it possible to debug it without any OS awareness. Normally you would need the OS awareness supplied by Simics Analyzer to debug user space programs.

14.2 Using GDB with Reverse Execution

gdb-remote supports an extension to the GDB remote protocol that allows the debugger to control the reverse execution functions in Simics. Simics comes with a prebuilt GDB-7.0 binary with reverse execution support for the most common target architectures and hosts.

Older versions of GDB does not know about this extension. There is, however, a patch for GDB 6.8 provided with Simics. This patch can be downloaded from `https://www.simics.net/pub/`. The patch can be used with standard GDB to compile GDB with reverse execution capabilities, see section 14.3. Note that the command names are different in this patch compared to GDB 7.0.

The reverse execution commands in GDB 7.0 are:

reverse-continue
: Run in reverse until a breakpoint or watchpoint is hit, or to the point where the reverse execution machinery runs out of history.

reverse-next
: Run in reverse and stop at the previous source code line. Will skip subfunction calls.

reverse-nexti
: Run in reverse and stop at the previous instruction. Will skip subfunction calls.

reverse-step
: Run in reverse and stop at the previous source code line. Will enter subfunction calls.

reverse-stepi
: Run in reverse and stop at the previous instruction. Will enter subfunction calls.

reverse-finish
: Run in reverse till the point where the current function is called.

Normal break- and watchpoints set with **break** and **watch** will also be triggered when running in reverse using **reverse-continue**

A small example of how to use **reverse-next**:

```
22          for (i = 0; i < 10; i++)
```

```
(gdb) p i
$2 = 0
(gdb) n
24            c = foo (i) + c;
(gdb) p i
$3 = 1
(gdb) reverse-next
22        for (i = 0; i < 10; i++)
(gdb) p i
$4 = 0
```

The amount of history that reverse execution keeps is limited; it is only possibly to reverse back to the point where reverse execution was enabled. If GDB recognizes that reverse execution has ran out of history, it will report an error. Note that this is not a fatal error, and the debugging session can continue, but without the possibility to reverse further than to the point where GDB reported the error.

```
(gdb) reverse-continue
Continuing.

No more history to reverse further.
_start () at start.c:17
17      {
```

14.3 Compiling GDB

If you do not want to (or cannot) use the GDB executable in *host*`/sys/bin/`, you will most likely have to compile GDB from source, even if your system already has GDB installed. The reason for this is that a given GDB executable is specialized both for the architecture of the computer you run it on (host), and the architecture of the computer that runs the programs you want to debug (target). Any GDB already installed on your computer will have target identical to host, but this is often not what you want when your target is a simulated machine.

The first step is to get the GDB source code. You can either get the unmodified GDB from `ftp://ftp.gnu.org/`, or a reverse execution-aware GDB from `https://www.simics.net/pub/`. In either case, the source will be packaged in a `.tar.gz` file.

The second step is to make sure you have all the tools necessary to compile GDB, such as GNU Make and a C compiler. On a Linux system, you probably have them already. On Windows, you will have to install Cygwin; get it at `http://www.cygwin.com/`.

That done, unpack and configure GDB like this:

```
~> tar zxfv gdb-6.8.tar.gz
~> cd gdb-6.8
~/gdb-6.8> ./configure --target=powerpc64-elf-linux
```

(On Windows, be sure to enter these commands in the **bash** shell installed as part of Cygwin.)

The **--target** flag to **configure** specifies which target architecture your new GDB binary will be specialized for (in this example, a 64-bit PowerPC). These flags are tabulated in the reference manual's section on **gdb-remote**, and will also be printed on the Simics console when you run **new-gdb-remote**.

```
~/gdb-6.8> make
```

The build process takes a while; when done, it will have left a gdb executable in the "gdb" subdirectory. You can execute it directly from that location:

```
~/gdb-6.8> ./gdb/gdb
```

Chapter 15

Using Simics with Other IDEs

Simics does not explicitly support IDEs other than those listed in this manual. If you would like to use Simics with such an IDE, there are a few options you could try:

- Use Simics's own user interface to control the debugging process (see chapter 10).
- If your IDE speaks the WDB (Wind River Debug) protocol, you can try using a **wdb-remote** Simics object to connect (see chapter 12).
- If your IDE speaks the GDB serial protocol, try connecting it directly to Simics with the help of a **gdb-remote** object (see chapter 14).
- If your IDE uses GDB as a debugger backend, try to make this GDB instance connect to Simics with **gdb-remote**.

Part IV

Performance

Chapter 16

Simulation Performance

This chapter covers various topics related to Simics performance and what can be done to measure and improve it. It discusses the general performance features provided by Simics Hindsight. For performance features provided by Simics Accelerator, see the *Accelerator User's Guide*.

Simics is a fast simulator utilizing various techniques such as run-time code generation to optimize performance. In some cases Simics can execute code faster than the target system being simulated, while it can also be considerably slower in other cases.

There are four major execution modes Simics uses to execute target instructions: hyper-simulation, VMP, JIT and interpreted mode.

Hyper-simulation means that Simics detects repetitive work performed by the target code and performs the effects of the code without actually having to run the code. In the most simple case this is a simple idle loop, but it can also be applied to more complex examples such as spin-locks and device polling. This is the fastest execution mode.

VMP, which is a part of Simics's x86 models, utilizes the virtualization capabilities of modern processors to run target instructions directly. This typically results in high simulation performance, but the host and target needs have the same instruction set, and you have to do special set up to enable it. VMP is currently only supported on x86 hosts.

JIT mode uses run-time code generation to translate blocks of the target instructions into blocks of host instructions. JIT mode is when Simics runs such translated blocks. This mode is supported by most target processor models in Simics.

Interpreted mode interprets the target instructions one by one. This mode is the slowest, but it is always available.

One of the largest differences in Simics performance depends on what Simics is used for, i.e., in which mode Simics runs. If Simics is started with the *-stall* option, for example to do cache modeling, the performance will decrease noticeably: with stall support Simics only uses interpreted mode, and internal caches have much lower granularity to enable cache simulation.

There are basically two ways to measure Simics performance:

- How fast the instructions are being simulated, typically measured in million target instructions per host second (MIPS).

- How fast the virtual times elapses.

In most cases the user is mostly interested in the first. Simics should execute instructions as fast as possible to finish the workload in shortest possible time. However, since Simics is a full system simulator, it is also important that the virtual time on the simulated machine advances quickly. That is important in cases where a program or operating system is waiting on a timer to expire or an interrupt from a device in order to proceed with the workload.

If we divide the wall-clock time on the host that Simics executes on, with the elapsed virtual time on the target machine, we get a slowdown number.

$$slowdown = Time_{host} / Time_{virtual}$$

A slowdown number of 2.3 means that Simics performance is 2.3 times slower than the system it simulates. A slowdown value of less than 1.0 means that Simics manages to execute the corresponding code faster than the system it simulates. The slowdown depends on various factors:

- The performance of the host which Simics runs on.
- The application which runs in Simics.
- The frequency of the target being simulated.
- The simulator time model.

The default time model in Simics is that each target instruction takes one target cycle to execute. That is the default, *Instructions Per Cycle* (IPC) is 1.0. This is a simplification (but in many cases an adequate approximation) compared to the actual time it takes on the real hardware to execute instructions. It is possible to change the IPC number using the command <**cpu**>**.set-step-rate**. For example:

```
simics> mpc8641d_simple.soc.cpu[0].set-step-rate ipc = 1.5
Setting step rate to 3/2 steps/cycle
simics> mpc8641d_simple.soc.cpu[0].set-step-rate ipc = 0.5
Setting step rate to 1/2 steps/cycle
```

In the first example, IPC of 1.5 means that Simics needs to execute 3 instructions for 2 cycles to elapse. In the second example, for each instruction executed two cycles elapse. Thus, with a lower IPC value, virtual time will progress faster and simulation slowdown will decrease.

Note that there is nothing wrong in changing the default IPC when it comes to the accuracy of the simulation. In many cases, the IPC observed for a given benchmark is much lower than the 1.0 that Simics assumes, and matching it will both make the simulation closer to the real hardware and improve the simulation speed, at least in virtual time. Simulations that profits most from this change are simulations involving devices and long memory latencies.

16.1 Measuring Performance

The **system-perfmeter** extension can be used to understand the performance you get out of Simics. The system-perfmeter is sample based, which means that you can see the performance during the workload execution, and how it varies, not only the end result when a workload is finished.

The easiest way to try it out is simply to issue the **system-perfmeter** command without any additional arguments:

```
simics> system-perfmeter
```

This will cause an sample to be taken every 1.0 virtual seconds. For each sample the system-perfmeter extracts various counters from Simics and displays the delta since last time. The output can look like this:

```
simics> c
SystemPerf: Total-vt Total-rt Sample-vt Sample-rt Slowdown  CPU Idle
SystemPerf: -------- -------- --------- --------- -------- ---- ----
SystemPerf:     1.0s     6.0s     1.00s     5.99s     5.99  97%   0%
SystemPerf:     2.0s     6.7s     1.00s     0.69s     0.69  97%   0%
SystemPerf:     3.0s     8.0s     1.00s     1.34s     1.34  92%   0%
SystemPerf:     4.0s     8.4s     1.00s     0.42s     0.42 100%   0%
SystemPerf:     5.0s     9.2s     1.00s     0.78s     0.78  98%  14%
SystemPerf:     6.0s    10.5s     1.00s     1.31s     1.31  96%  55%
SystemPerf:     7.0s    10.7s     1.00s     0.12s     0.12  92%  93%
SystemPerf:     8.0s    10.7s     1.00s     0.00s     0.00 100% 100%
```

Here we can see the execution for the first 8 virtual seconds and the corresponding performance measured in each second sample. To simulate these 8 virtual seconds, it took Simics 10.7 host seconds, thus the average slowdown is 1.34.

The `CPU` column shows how much percent of the host CPU that Simics has used, allowing you to notice if there is another process consuming the host CPU resource. Another reason for CPU utilization to be low can be that Simics itself is running in real-time mode where Simics sleeps so that virtual time does not race ahead of host time.

When an 'idle' condition has been detected (see chapter 16.4.1), the total idleness of the system is reported in the `Idle` column. If the simulated system consists of multiple processors and you wish to see how much each processor is idling you can use the *-cpu-idle* switch to **system-perfmeter**. Note that idling is defined by the simulator, not by the target architecture (see chapter 16.4.1). With the *-cpu-exec-mode*, information is also gathered and printed on how simulation steps are executed in the CPU model. The fastest mode to be executing in is idle, followed by VMP, JIT, and interpreter.

The **system-perfmeter** can also be used to get an understanding of which processor that takes the longest time to simulate. The *-cpu-host-ticks* switch adds extra columns per CPU for this. For example:

```
SystemPerf: Total-vt Total-rt Sample-vt Sample-rt Slowdown  CPU Idle [   0   1   2   3 ]
```

```
SystemPerf: -------- -------- --------- --------- -------- ---- ----   --- --- --- ---
SystemPerf:     1.0s    15.6s     1.00s    15.63s     15.6  99%  74% [  71  11  11   8 ]
SystemPerf:     2.0s    32.6s     1.00s    16.97s     17.0  98%  72% [  70  11  10   9 ]
```

Here we have a 4 CPU system which is idle roughly 70% and the last columns shows that CPU0 takes 70% of the time to simulate, while the other three about 10% each. CPU0 is working while the other CPUs are idling.

The *-module-profile* flag enables live profiling of the main Simics thread. The profiling is sample based, and any sample hitting in code produced dynamically by a JIT engine will be reported as "*classname* JIT". The *-module-profile* data is not printed in the standard line print mode, so you must use either *-top* or *-summary* to get profiling information.

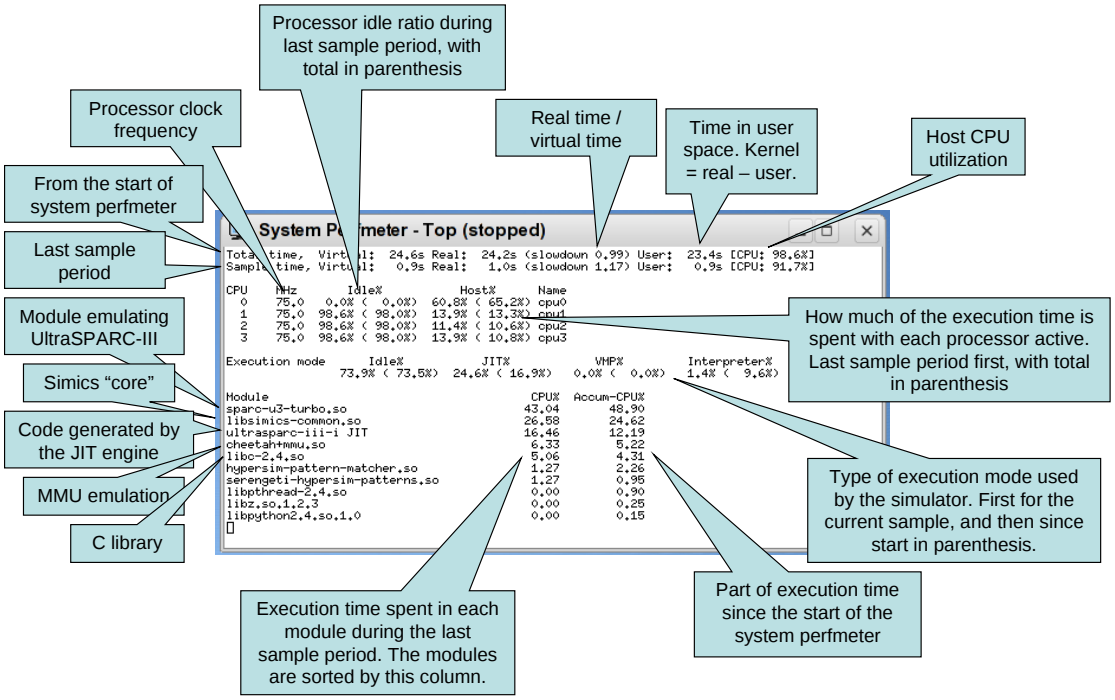

Figure 16.1: Annotated output from **system-perfmeter *-top -module-profile -cpu-exec-mode***

Figure 16.1 explains the output of **system-perfmeter**. There are many other options to the system-perfmeter command, please read the associated help text for details.

```
simics> help system-perfmeter
```

16.2 Multithreaded Simulation Profiling

With Simics Accelerator, simulation performance can be increased by utilizing host system parallelism. Another dimension is added which affects Simics performance: the amount of available target system parallelism greatly impacts overall simulation performance.

Multithreading in Simics is based upon the cell concept. A cell ideally contains tightly coupled parts of the target system (typically one or more CPUs and associated devices). Different cells can be simulated in parallel, but a single cell will always be simulated in a single-threaded fashion (see the *Accelerator User's Guide* for more details).

The Multithreaded Simulation Profiler tool (**mtprof**) can be used to analyze some performance aspects of multithreaded simulation. Mtprof is primarily useful to

- detect parallelism bottlenecks
- predict how fast the simulation would run on a more parallel host machine
- give some insights into the performance impact of various latency setting.

The Multithreaded Simulation Profiler is started with the following command:

```
simics> enable-mtprof
```

When mtprof is enabled, Simics keeps track of how much CPU time is spent simulating each cell in the system. Once Simics has been running for a while, it is possible to ask mtprof for an overview:

```
simics> mtprof.cellstat
=============================================================
  cellname                    rt    %elapsed  %cputime
-------------------------------------------------------------
  ebony3.cell3             77.0s      93.6%     80.0%
  ebony1.cell1              8.9s      10.9%      9.3%
  ebony2.cell2              8.2s      10.0%      8.5%
  ebony0.cell0              2.1s       2.5%      2.1%
-------------------------------------------------------------
  elapsed_realtime         82.2s     100.0%     85.5%
=============================================================
```

From the above output, we can conclude that cell3 is a limiting factor: the other cells frequently have to wait upon this cell in order to keep the virtual time in sync. Investigating why cell3 is so expensive to simulate is the next natural step: there might be an expensive poll loop or the idle optimization might not function properly, for instance. Other potential ways to address the load imbalance include

- splitting cell3 into multiple cells (if it contains multiple CPUs)
- changing the step rate of CPUs belonging to cell3.

The mtprof tool can also be used to estimate how fast the simulation would run on a machine with enough host cores and Simics Accelerator licenses to allow Simics to assign a dedicated host core to each cell:

```
simics> mtprof.modelstat
=============================================================
  latency      rt_model  realtime/rt_model
-------------------------------------------------------------
    10 ms         81.9s       100.4%
    40 ms         81.2s       101.2%
   160 ms         80.3s       102.4%
   640 ms         78.6s       104.6%
=============================================================
```

The latency column corresponds to various **set-min-latency**. In this case, the simulation was run with a 10 ms latency, which means that the model predicts that the simulation would take 81.9 s to run (which is pretty close to the measured value of 82.2 s above).

It is important to note that the performance model does not take modified target behavior due to different latency settings into account (which can be a huge factor if the cells interact). With substantial inter-cell interaction, only the row corresponding to the current latency setting should be trusted.

Below is an example from a more evenly loaded system run with a min-latency of 1 ms:

```
simics> mtprof.modelstat
=============================================================
  latency      rt_model  realtime/rt_model
-------------------------------------------------------------
     1 ms         35.3s       107.4%
     4 ms         29.5s       128.8%
    16 ms         25.9s       146.3%
    64 ms         24.2s       156.5%
=============================================================
```

In this case, we see that increasing the latency setting to about 16 ms would improve simulation performance substantially (once again, without taking changed target behavior into account).

Quite often, the target behavior is not static but varies with simulated time. In that case, it is often useful to export the collected data and plot it using an external tool:

```
simics> mtprof.save-data output.txt
```

The exported data is essentially the information provided by **mtprof.cellstat** and/or **mtprof.modelstat**, but expressed as a function of virtual time (exported in a plot friendly

way). The **mtprof.save-data** command takes multiple flags which can be used to customize the output. One useful flag is *-oplot*:

```
simics> mtprof.save-data mtprof-plot.m -oplot
```

which outputs the data in the form of an Octave file together with commands which plots the data.

16.3 Platform Effects

Starting with the most important, consider these factors when choosing a platform to run Simics on:

CPU Speed
Simulation is a very compute intensive application. As a general rule, a machine with higher compute performance will outperform a lower compute performance machine unless the simulation is starved for memory. The SPECint2006 benchmark suite (`www.spec.org`) is a good indicator of compute performance and can be used to compare systems with different processors. Both processor architecture and clock frequency affects the performance. Within an architecture performance scales almost linearly with clock frequency.

Other CPU features such as the size of the caches does affect performance, but less so than the architecture and the clock frequency.

Memory Size
The simulator can operate with less memory than the simulated memory size. It does that by allocating pages only when used and swapping out pages to disk when running low on memory. The memory size needs to be large enough so that the paging of memory to disk does not hurt performance too much.

Number of Cores
Simics Accelerator will utilize multiple host threads to simulate cells in parallel. If all cells consume equal amount of time to simulate, then the simulator can utilize as many host cores as there are cells. In practice, the workload is always skewed meaning that fewer host cores can be effectively utilized. It is more efficient to use multiple cores or processors on a single machine compared to distributing the simulation across multiple host machines.

Operating System
The 64-bit versions of Simics are faster than the 32-bit versions. This is especially true when the target system includes 64-bit processors or if the system has a large memory footprint.

File System
Running from local disk is faster than running over a network file system. When running low on memory, make sure that the swap directory used by the memory limit feature is located on local disk.

16.4 Workload Characteristics

The performance of Simics sometimes depend on the kind of software that runs inside the target machine being simulated. Thus, the slowdown can vary a lot depending on what the target software is currently executing. Here are some general tips for understanding what decreases simulation speed.

Floating-point intense workloads
: If the target software runs floating-point arithmetic instructions frequently, the performance is likely lower compared to running integer based workloads.

Supervisor code
: If a workload runs much of the code in supervisor mode such as frequently causing exceptions, this type of code normally runs slower than regular user-level code.

I/O workloads
: If the code does frequent accesses to devices, compared to accessing RAM, Simics needs to do more work to simulate this, which slows down simulation. Note that this type of workload are an excellent match for a lower IPC setting.

Memory usage
: If the target software is using more memory than available on the host. This is typically a problem when the simulated machine has equal or more memory than the host it executes on **and** the software also uses it. This will cause Simics to swap out pages on disk, which decreases performance.

Event usage
: Simics modules which frequently post events with short time quanta. For example, if a device posts an event each 10 cycles to keep a counter register updated this will severely affect performance.

Typically, target code which runs with low performance on real hardware due to bad cache behavior, bad memory locality etc. will also cause Simics to run with poor performance.

16.4.1 Idle Loops and Performance

When an operating system does not have any processes to schedule it typically runs some tight loop waiting for an interrupt to occur. This is referred to as the *idle loop*. The way the idle loop is implemented varies between operating systems and the capabilities in the underlying hardware.

For example, the most simple idle loop would be a "branch to itself" instruction. When the processor reaches this instruction, nothing but an interrupt will cause the execution to proceed somewhere else. Another example is when the operating system uses some kind of power-down mode on the processor, causing the processor to stop executing any more instructions (and consequently consume less power). Some processors also have dedicated instructions causing the processor to stop until something interesting happens, such as the x86 `HLT` instruction. Processor idling in Simics is defined by what the simulator can detect

and usually includes architectural states such as halt or power-down, but can also be loops normally executed by the processor.

A fast simulation of the idle loop, is very important in some cases. For example, when simulating multiple processors, we want to use as much of the host CPU cycles as possible for simulating the processor that actually performs useful tasks. Rather than wasting cycles on the idle loop.

Even when simulating a single CPU, fast idle loops can be important, since all of the active processes might be stalling on disk or some other peripheral. Execution of the processes will not continue unless, for example a SCSI disk issues an interrupt.

There are three ways in which Simics optimizes idle loops:

1. Automatic hyper-simulation detected by the processor model itself.
2. Idle loop optimizers using the hyper-simulation framework.
3. Tailor-made idle loop optimizing module.

Simics processor models can sometimes detect idle conditions. When the processor model detects a branch to itself, there is no point in simulating the instruction, if it branches to itself repeatedly.

Instead, Simics is capable of fast-forwarding time until an event that can generate an interrupt is about to be executed. Hence, this model is equivalent to running the branch millions of times. However, it is much faster.

In some cases the idle loop in the operating system is more than an single instruction, it might be a loop checking a variable in memory for the next process to schedule, or something similar. To handle these more difficult cases a tailor-made Simics extension might be needed, or it can be handled with hyper-simulation (see chapter 16.5).

Simics distributions sometimes include dedicated tailor-made idle-loop optimizers, such as **v9-sol9-idle-opt** (a Solaris idle loop optimizer for SPARC-V9 processors).

It is generally possible to implement idle loop optimizers within the hyper- simulation framework.

16.5 Hyper-simulation

The term *hyper-simulation* refers to a simulator feature which can detect, analyze and understand, frequently executed target instructions and fast-forward the simulation of these. Thus providing the corresponding results more rapidly.

Being able to detect the idle loop (see chapter 16.4.1) is one example of when this technique is applicable. A much more extreme hyper-simulation task would be to understand a complete program and simply provide the corresponding result without actually starting the program. Naturally, this is hardly ever applicable. Busy-wait loops and spin-locks are more realistic examples of cases where it is easy to optimize away the execution with hyper-simulation.

The following cases are automatically hyper-simulated:

Target	Instruction	Comment
ARM	`mcr`	Enabling “Wait for Interrupt”
MIPS	`wait`	
PowerPC	`mtmsr`	Setting `MSR[POW]`.
PowerPC	`b 0`	Branch to itself
x86	`hlt`	

Hyper-simulation should be as non-intrusive as possible, the only difference that should be noticeable as a Simics user is the increased performance. Registers, timing, memory contents, exceptions, interrupts etc. should be identical.

Advanced hyper-simulation may have some intrusions regarding Simics features:

- Device and memory access count will be too low if accesses are optimized away.
- Breakpoints inside a hypersim detected code segment will not trigger every time.
- Breakpoints on accesses to memory or devices will not hit every time since many of these accesses can have been optimized away.

Advanced fast-forwarding is activated by default when running in normal (non-stall) mode. The command **enable-hypersim** activates the advanced fast-forwarding. To deactivate this feature the **disable-hypersim** command can be used. The **hypersim-status** command gives some details on what hypersim features that are currently active.

Hypersim patterns are typically fragile, since they depend on an exact instruction pattern. Simply changing the compiler revision or an optimizing flag to the compiler can break the pattern from being recognized anymore.

The *MPC8641-Simple* machine does not use hypersim patterns, but if you have access to the *Ebony* you can try the following examples. The example uses the `ebony-linux-common.simics` script:

```
> ./simics targets/ebony/ebony-linux-common.simics

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> disable-hypersim
simics> system-perfmeter -realtime -mips
Using real time sample slice of 1.000000s
simics> c
SystemPerf: Total-vt Total-rt Sample-vt Sample-rt Slowdown  CPU Idle  MIPS
SystemPerf: -------- -------- --------- --------- -------- ---- ---- -----
SystemPerf:     0.1s     0.3s     0.09s     0.33s      3.4 100%   0%    29
SystemPerf:     0.7s     1.3s     0.56s     1.00s      1.8  97%   0%    55
SystemPerf:     0.8s     2.3s     0.13s     1.00s      7.6  99%   0%    13
SystemPerf:     2.0s     3.3s     1.22s     1.00s      0.8  95%   0%   122
SystemPerf:     4.2s     4.3s     2.24s     1.00s      0.4  78%   0%   223
SystemPerf:     5.8s     5.3s     1.54s     1.00s      0.6  97%   0%   153
SystemPerf:    11.3s     6.3s     5.46s     1.00s      0.2  99%   0%   543
SystemPerf:    15.9s     7.3s     4.65s     1.00s      0.2  98%   0%   462
SystemPerf:    21.7s     8.3s     5.82s     1.00s      0.2  99%   0%   579
SystemPerf:    27.5s     9.3s     5.82s     1.00s      0.2 100%   0%   579
SystemPerf:    33.3s    10.3s     5.80s     1.00s      0.2  99%   0%   579
[cpu0] v:0xc0003d24 p:0x000003d24  beq- cr7,0xc0003d0c
```

```
simics> enable-hypersim
simics> c
SystemPerf:    65.6s    11.2s    32.23s     0.88s      0.0  98%  85%  3673
SystemPerf:   491.1s    12.2s   425.52s     1.00s      0.0 100% 100% 42382
SystemPerf:   908.4s    13.2s   417.36s     1.00s      0.0  99% 100% 41550
SystemPerf:  1305.9s    14.2s   397.44s     1.00s      0.0 100% 100% 39745
SystemPerf:  1746.3s    15.2s   440.44s     1.00s      0.0  99% 100% 44039
SystemPerf:  2200.9s    16.2s   454.59s     1.00s      0.0  99% 100% 45457
[cpu0] v:0xc0003d0c p:0x000003d0c  beq- cr4,0xc0003d1c
simics>
```

This configuration has an linux idle loop optimizer by default. We start by disabling hypersim and execute the code "normally" during the boot. After 6 seconds (host) or 12 seconds (virtual) the boot is finished and the operating system starts executing the idle loop. The idle loop itself is executed quickly in Simics, running at 579 MIPS. When idling, almost 6 virtual seconds is executed for each host second. That is, Simics executes 6 times faster than the hardware (the processor is configured to be running at 100 MHz).

Next, execution is stopped and hypersim is enabled. Now we can see the idle loop optimizer kicking in. Now 400 virtual seconds is executed each host second, or about 70 times faster than without hypersim enabled.

16.6 VMP

The VMP add-on for Simics makes use of hardware virtualization support to provide vastly improved performance when simulating x86-based systems. It is an optional part of the x86-based models.

The VMP feature requires that the host machine running Simics implements the Intel ® VT feature set. That means that the host machine must have a processor from this list:

- Intel ® Core™ Duo / Solo processor
- Intel ® Pentium ® D 920/930/940/950 processor
- Intel ® Pentium ® 4 662/672 processor
- Intel ® Core™2 Duo / Quad / Extreme processor
- Intel ® Xeon ® 30xx / 31xx / 32xx / 33xx / 51xx / 52xx / 53xx / 54xx / 55xx / 72xx / 73xx / 74xx processor
- Intel ® Core™ i7 processor

The virtualization feature must also be enabled in the host machine firmware. Look for options under either *Security* or *Virtualization* to find where to enable Intel ® VT in your firmware.

All major features of Simics, including full inspectability of simulated state, ability to model heterogeneous systems, and reverse execution, are fully supported when running with VMP.

16.6.1 Installing the VMP kernel module

Change directory to the Simics Base package and run:

```
joe@computer$ ./scripts/vmp-kernel-install.sh
```

The script will compile and install the kernel modules that are used by VMP. The script needs to be run after every reboot (unless you setup the system to load the kernel modules automatically).

16.6.2 Running with the VMP add-on

With the VMP kernel modules installed, VMP will be enabled by default for each processor. You can disable VMP by running the **disable-vmp** command.

Due to details about how the Intel ® VT feature that VMP is based on works, the acceleration may not kick in. Use the system-perfmeter to find out if a processor actually uses the VMP execution mode. To find out why VMP is not used, either raise the log level of the CPU in question or use the **info** command on the CPU.

16.6.3 Current limitations of the VMP add-on

The VMP packages are currently only available on 32-bit and 64-bit Linux host systems. Windows is currently not supported as a host. Note that virtualizers such as VMWare or Xen do not expose the Intel ® VT feature, meaning that VMP needs to run in a non-virtualized environment.

16.7 Performance Tweaks

There are a number of parameters in Simics which can be tweaked which might lead to increased performance.

- The instruction per cycle (IPC) parameter can be decreased, see command <**cpu**>**.set-step-rate**. Similar to reducing the CPU frequency, this will cause virtual time to progress more rapidly with the same amount of instructions executed.
- Devices sometimes have a timing model that can be changed by attributes.
- When real-time performance is required, the **real-time** module provides some means to achieve this. See command **enable-real-time-mode**
- For distributed simulation and multithreading, you might want to tweak the default latency between the simulation cells. Increasing the latency will diminish the cost of synchronizing the simulation. More information is available in the *Simics Accelerator User's Guide* document.
- In a similar way, you may want to check the time quantum used to schedule the simulated processors within a Simics process or a simulation cell. Processors unrelated to each other (they do not share memory, for example) do not need to be simulated with a small time quantum.

- The frequencies of the processors can be lowered (*cpu_freq_mhz*). This will cause virtual time to progress more rapidly with the same amount of instructions executed.
- If page/swap activities can be monitored on the host running Simics you might want to decrease the memory-limit, see the **set-memory-limit** command. See chapter 7.1.2.

Index