

WIND RIVER

# Wind River® Simics® Analyzer

USER'S GUIDE

4.6

<i>Revision</i>	4081
<i>Date</i>	2012-11-16

Copyright © 2010–2012 Wind River Systems, Inc.

All rights reserved. No part of this publication may be reproduced or transmitted in any form or by any means without the prior written permission of Wind River Systems, Inc.

Wind River, Simics, Tornado, and VxWorks are registered trademarks of Wind River Systems, Inc. The Wind River logo is a trademark of Wind River Systems, Inc. Any third-party trademarks referenced are the property of their respective owners. For further information regarding Wind River trademarks, please see:  
[www.windriver.com/company/terms/trademark.html](http://www.windriver.com/company/terms/trademark.html)

This product may include software licensed to Wind River by third parties. Relevant notices (if any) are provided in your product installation at the following location:  
`installDir/LICENSES-THIRD-PARTY/`.

Wind River may refer to third-party documentation by listing publications or providing links to third-party Web sites for informational purposes. Wind River accepts no responsibility for the information provided in such third-party documentation.

#### **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](http://www.windriver.com)

For information on how to contact Customer Support, see:  
[www.windriver.com/support](http://www.windriver.com/support)

# Contents

<b>1</b>	<b>Introduction</b>	<b>7</b>
<b>2</b>	<b>Target Software Tracking</b>	<b>9</b>
2.1	Availability . . . . .	9
2.1.1	Keyword explanation . . . . .	9
2.1.2	Linux tracker availability . . . . .	10
2.1.3	VxWorks tracker availability . . . . .	11
2.1.4	Wind River Hypervisor tracker availability . . . . .	11
2.1.5	OSE tracker availability . . . . .	12
2.1.6	QNX tracker availability . . . . .	12
2.1.7	CPU mode tracker availability . . . . .	12
2.2	Components and Commands . . . . .	12
2.3	Tracking Processes . . . . .	14
2.4	Configuration . . . . .	14
2.4.1	Configuration Linux . . . . .	14
2.4.2	Configuration VxWorks . . . . .	15
2.4.3	Configuration Wind River Hypervisor . . . . .	16
2.4.4	Configuration OSE . . . . .	18
2.4.5	Configuration QNX . . . . .	18
2.4.6	Configuration Partition . . . . .	19
2.5	Software Domains . . . . .	20
2.6	Limitations . . . . .	20
<b>3</b>	<b>Debugging Target Code</b>	<b>22</b>
3.1	Debugging Environment Tour . . . . .	22
3.1.1	Debug Perspective . . . . .	22
3.1.2	Views . . . . .	23
3.1.3	Debug Configuration . . . . .	24
3.2	Getting Started with a Debug Session . . . . .	24
3.2.1	Starting Simics . . . . .	24
3.2.2	Attaching the debugger to a Simics Session . . . . .	25
3.3	Setting Up Symbol Information . . . . .	25
3.3.1	Adding a symbol file . . . . .	26
3.3.2	Re-mapping a segment or sections . . . . .	27
3.3.3	Browsing symbols . . . . .	28

3.3.4	Symbol Browser Limitations	28
3.3.5	About Symbolic Debugging	29
3.4	Setting Up Source Code Path Mappings	30
3.5	Setting Breakpoints	31
3.5.1	Function Breakpoints	31
3.5.2	Line Breakpoints	31
3.5.3	Watchpoints	31
3.5.4	Context Creation	32
3.5.5	Context Destruction	32
3.5.6	On Physical Address	33
3.5.7	Ignore Count	33
3.5.8	Manipulating Breakpoints	34
3.5.9	Status	34
3.5.10	Scoping	35
3.5.11	Breakpoint Icons	35
3.6	Using Context Queries	36
3.6.1	About Context Queries	36
3.6.2	Context Query Syntax	37
3.6.3	Examples	37
3.6.4	Difference in the Command Line	38
3.7	Expressions	38
3.7.1	Populating the Expressions View	38
3.7.2	Adding Expressions	39
3.7.3	Deleting Expressions	39
3.8	The Command Line Debugger	39
3.8.1	Debug Contexts	40
3.8.2	Configuration	42
3.8.3	Breakpoints	43
3.9	Debugger Limitations	44
<b>4</b>	<b>Debugging</b>	<b>46</b>
4.1	Contexts	46
4.2	Symbol Tables	46
4.3	Using Simics Contexts with GDB	47
<b>5</b>	<b>Code Coverage</b>	<b>48</b>
5.1	Requirements	48
5.2	Limitations	48
5.3	The code-coverage Command	49
5.4	Saving Manually	50
<b>6</b>	<b>Profiling Tools</b>	<b>53</b>
6.1	Instruction Profiling	53
6.1.1	Virtual Instruction Profiling	55
6.2	Data Profiling	55
6.3	Examining the Profile	56

<b>7</b>	<b>Cache Simulation</b>	<b>59</b>
7.1	Introduction to Cache Simulation with Simics . . . . .	59
7.2	Simulating a Simple Cache . . . . .	60
7.3	A More Complex Cache System . . . . .	61
7.4	Caches Models and Instruction Fetches . . . . .	65
7.5	Workload Positioning and Cache Models . . . . .	65
7.6	Using g-cache . . . . .	66
7.7	Understanding g-cache Statistics . . . . .	67
7.8	Speeding up g-cache simulation . . . . .	68
7.9	Cache Miss Profiling . . . . .	68
7.10	Using g-cache with Several Processors . . . . .	71
7.11	Simulating Private Caches . . . . .	71
7.12	g-cache Limitations . . . . .	73
<b>8</b>	<b>Processor-specific Features and Limitations</b>	<b>74</b>
<b>9</b>	<b>OS Awareness Details</b>	<b>76</b>
9.1	Tracker Objects And System Configurations . . . . .	76
9.2	Tracker Activation . . . . .	78
9.3	The Node Tree . . . . .	78
9.4	The Software Interface . . . . .	80
9.4.1	software_interface_t . . . . .	80
9.5	Using Context Objects . . . . .	83
9.6	System Call Notifications . . . . .	84
9.7	Node path patterns . . . . .	84
9.7.1	Introduction . . . . .	84
9.7.2	Summary . . . . .	86
9.7.3	Limitations . . . . .	86
9.8	Configuring the Software Tracker . . . . .	86
9.8.1	Configuration With Parameter Files . . . . .	87
9.8.2	Configuration With Attributes . . . . .	87
9.8.3	Configuration With Interface Calls . . . . .	87
9.8.4	List of Tracker Modules . . . . .	88
	CPU Mode Tracker . . . . .	88
	Linux Tracker . . . . .	88
	Wind River Hypervisor . . . . .	90
	VxWorks Tracker . . . . .	91
	QNX Tracker . . . . .	93
	OSE Tracker . . . . .	93
	Partition Tracker . . . . .	94
9.9	Using the Software Tracker in Scripts . . . . .	95
<b>10</b>	<b>Graphical Timeline Example</b>	<b>98</b>
10.1	Analysis of the Firststeps Linux boot . . . . .	98
10.2	Analyzing of a simple userspace program . . . . .	99



# Chapter 1

## Introduction

Simics Analyzer provides features for analysis and debugging of software applications within Simics. It makes it possible to see what software processes are running, and to inspect, analyze and debug them individually. It includes tools for doing non-intrusive code coverage analysis and other profiling. This document describes how these features work, and shows how to use them.

Many of the features described in this documentation build on features that are part of Simics Hindsight, and it is recommended that you read the *Hindsight User's Guide* to familiarize yourself with the debugging tools of Simics.

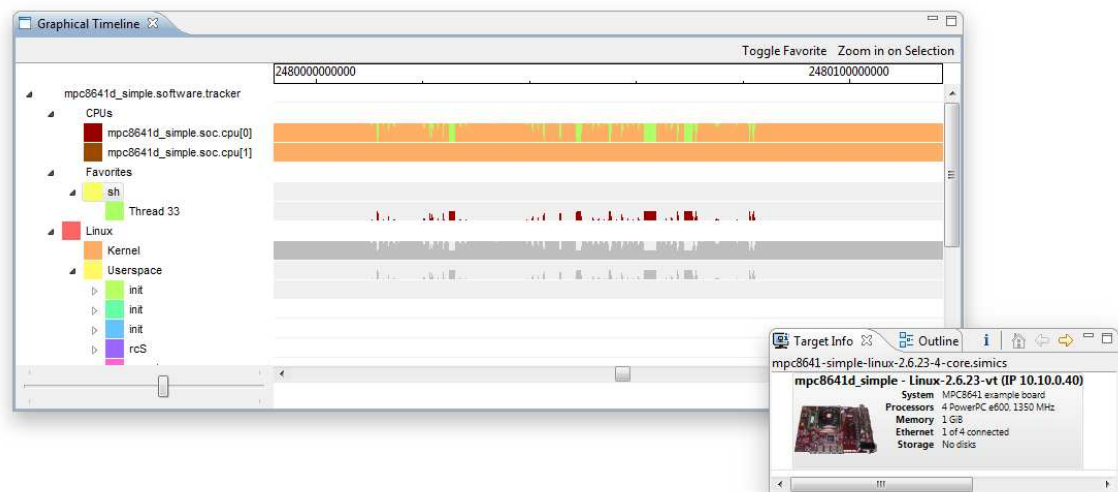


Figure 1.1: The Analyzer views

Part of Simics Analyzer is provided as views for Eclipse. These views provides an overview of what runs on the system and what has run when and where. There are two major views: a graphical timeline view which shows what has run when and where during a simulation, and a view showing the current software and where it is currently running.

These views are used in combination with the rest of Simics Eclipse. They are documented in the *Eclipse User's Guide*.

Simics Analyzer also includes a beta version of a new Eclipse based debugger. See *Debugging with Eclipse* (only available in PDF-format) for documentation about how to install and use it.



## Chapter 2

# Target Software Tracking

Simics can be very helpful in understanding how software runs on a target system. This is true for user-level applications running as processes on the operating system, as well as for operating system kernel tasks, system calls and hardware drivers. In many modern systems, there are even more layers of software, with a hypervisor running several operating system kernels, which are running a number of user tasks.

To be able to debug, analyze, or otherwise track these tasks and processes, Simics employs a number of *OS awareness* modules that examines the state of the target system to determine when and where each task is running. It does this by for example examining registers and memory contents and tracking changes in these and capturing exceptions and CPU mode transitions. It does not require any kind of modification or instrumentation of the software it is tracking.

### 2.1 Availability

Simics OS awareness has built-in support for a variety of different operating systems, including hypervisor support. However, not all platforms, versions and features are supported by all trackers. This section gives an overview over systems that are known to work. Depending on the system that should be monitored, there are different system features that are of interest. Many features are shared between all trackers, and some are only available by some trackers. Some features only exists for some OSes.

#### 2.1.1 Keyword explanation

This section briefly explains the keywords used in this chapter to describe features supported or unsupported by various trackers.

##### **Thread**

Indicates if the tracker supports tracking individual threads

##### **Processes**

Indicates if the tracker supports tracking user-space applications

**SMP**

Indicates if the tracker supports an SMP kernel

**UP**

Indicates if the tracker supports a UniProcessor kernel

**Arch**

Lists the architectures for which trackers exist

**RTP**

Indicates if the tracker support tracking Real Time Processes (VxWorks only)

**Version**

Indicates the operating system version, or distribution version known to work

**Idle**

Indicates if the tracker supports reporting when the operating system is idle

**Supervisor mode**

Indicates if the tracker supports reporting when the processor executes in supervisor mode

**User mode**

Indicates if the tracker supports reporting when the processor executes in user mode

### 2.1.2 Linux tracker availability

The various Linux configurations that are known to work with the Linux tracker can be found in this table.

Arch	Version	Processes	SMP	Threads	UP
arm32	2.6	X		X	X
ppc32	2.4	X	X	X	X
ppc32	2.6	X	X	X	X
ppc32	Wind River Linux 3.0	X	X	X	X
ppc32	Wind River Linux 4.0	X	X	X	X
ppc64	2.4	X	X	X	X
ppc64	2.6	X	X	X	X
ppc64	Wind River Linux 3.0	X	X	X	X
ppc64	Wind River Linux 4.0	X	X	X	X
x86	2.4	X	X	X	X
x86	2.6	X	X	X	X
x86	Wind River Linux 3.0	X	X	X	X
x86	Wind River Linux 4.0	X	X	X	X
x86-64	2.4	X	X	X	X
x86-64	2.6	X	X	X	X
x86-64	Wind River Linux 3.0	X	X	X	X
x86-64	Wind River Linux 4.0	X	X	X	X

### 2.1.3 VxWorks tracker availability

The various VxWorks configurations that are known to work with the VxWorks tracker can be found in this table.

Arch	Version	Idle	RTP	SMP	Tasks	UP
arm32	6.8	X	X	X	X	X
arm32	6.9	X	X	X	X	X
ppc32	5.5	X			X	X
ppc32	6.7	X	X	X	X	X
ppc32	6.8	X	X	X	X	X
x86	6.7	X	X	X	X	X
x86	6.8	X	X	X	X	X

### 2.1.4 Wind River Hypervisor tracker availability

The various Wind River Hypervisor configurations that are known to work with the Wind River Hyper Visor tracker can be found in this table.

Arch	Version	Contexts	Linux SMP	Linux UP	VxWorks
arm	1.3	X			X
ppc32	1.0	X		X	X
ppc32	1.1	X		X	X
ppc32	1.2	X	X	X	X
ppc32	1.3	X	X	X	X
x86-64	1.0	X		X	X
x86-64	1.1	X		X	X

### 2.1.5 OSE tracker availability

The various OSE configurations that are known to work with the OSE tracker can be found in this table.

Arch	Version	Processes	UP
ppc32	5.3	X	X
ppc32	5.4	X	X
ppc32	5.5	X	X

### 2.1.6 QNX tracker availability

The various QNX configurations that are known to work with the QNX tracker can be found in this table.

Arch	Version	Processes	SMP	UP
ppc32	6.4	X	X	X

### 2.1.7 CPU mode tracker availability

The CPU mode tracker a very simple tracker, that has the properties found in this table.

Arch	Version	Supervisor mode	User mode
All	N/A	X	X

## 2.2 Components and Commands

To be able to use the features described in this chapter, the configuration scripts for the target system needs to support the OS awareness functionality. In a supported system configuration, there is a slot called `software` in the system component. This can be used to control the software tracking. Furthermore, to be able to track the software, it needs to be configured

with software tracker settings for the software you are running. This is often included in the standard configuration scripts. To see which tracker configuration is used, you can use the **info** command on the software slot:

```
simics> mpc8641d_simple.software.info
Information about mpc8641d_simple.software [class os_awareness]
=====

Software:
  Tracker : linux_tracker
    CPUs  : mpc8641d_simple.soc.cpu[0]
            mpc8641d_simple.soc.cpu[1]

Slots:
  tracker : mpc8641d_simple.software.tracker

Connectors:
```

In the example above, the software tracker is configured with the *linux\_tracker* module. If there is no configuration, *none* is shown.

If the configuration scripts have not set up any software tracker configuration, and there are tracker modules supporting the software you are running, you can extend the configuration with this information. See section 9.8 for more information about how to do this.

As a first example of how the tracker can be used, there is a **list** command that lists the processes being tracked. The output depends on the tracker module in use. This shows an example for the Linux tracker, with output resembling that of the *ps* command. Remember to boot the system before trying this:

```
simics> mpc8641d_simple.software.list

Process      Binary      PID  TID
...
init          1          1
telnetd       971        971
httpd         973        973
sh            /bin/sh    974        974
```

The tracker is not enabled by default, since enabling it will reduce performance somewhat. It will automatically be enabled when you start using it, as seen in the example above. It can also be enabled manually

```
simics> mpc8641d_simple.software.enable-tracker
```

And when it is no longer needed, it can be disabled again.

```
simics> mpc8641d_simple.software.disable-tracker
```

## 2.3 Tracking Processes

To be able to debug a process running on the target system, Simics needs to be able to match the process with a **context** object in Simics. The context object handles things like breakpoints in the virtual address space and source line information. A context can be created and managed manually, but it is much more convenient to let the OS awareness module do it automatically. The **track** is used to set up new processes with contexts.

```
simics> mpc8641d_simple.software.track myprog
```

This will instruct the process tracker to watch for new processes with the name *myprog*, and when such a process is started, a context object is attached to it so that whenever a process executes code in that process, the context object will be active. This means that you can do things like run until some processor core starts executing code in the process:

```
simics> myprog0.run-until-active
```

In this example, *myprog0* is the name of the context object that was created by the **track** command. When scripting Simics, however, it is much better to store the returned context object from the **track** command in a variable and use that in the script:

```
# Here, $system is a variable naming the top-level component
$ctx = ($system.software.track myprog)
$ctx.run-until-activated
```

By using the low-level interfaces described in chapter 9, it is possible to attach contexts to other levels of the software abstractions, such as a single thread or kernel space execution.

## 2.4 Configuration

This section gives an overview on how to configure various software trackers using the built-in detect commands. For more detailed information about tracker configuration see chapter 9. For more detailed information about each command used in this section see *Simics Reference Manual* or the built-in help.

### 2.4.1 Configuration Linux

The configuration of a Linux tracker can be divided into two parts. The first part, which is only meant to be done for a new kernel, or sometimes in order to enable new tracker features, is to generate a configuration file. This configuration file contains information about the running Linux operating system, required by the tracker. Generation of this parameters file

is done by the **linux-autodetect-settings** command. In order to get a successful detection of parameters, Linux has to be up and running. The generated parameters file can then be loaded into the OS awareness framework with the **load-parameters** command. The tracker is now ready for use. For the detection to be successful on x86 systems with kernels 2.6.32 or higher a symbol file is needed as well. For more information use the built-in help for the **linux-autodetect-settings** command.

Below is an example of how to configure and enable the Linux tracker, remember to boot the system first.

```
simics> ebony.software.linux-autodetect-settings param-file = ebony-linux.params
[ebony.software.tracker info] enabling the tracker
[ebony.software.tracker info] Autodetecting using 1 of 1 processor(s)
Saved autodetected parameters to ebony-linux.params
[ebony.software.tracker info] disabling the tracker
```

```
simics> ebony.software.load-parameters ebony-linux.params
```

## 2.4.2 Configuration VxWorks

The configuration of the VxWorks tracker can be divided into two parts. Where the first part is only meant to be done the first time the system is configured, the VxWorks image has been replaced, or to enable new features provided by the tracker framework that requires a reconfiguration. During this phase the tracker framework analyzes the VxWorks image and extracts important data required by the tracker to monitor the system and generates a parameters file. The second part is simply to load this parameters file into the OS awareness system. For VxWorks 5.5 kernels, the **vxworks-detect-settings** command requires the system image to be loaded into memory. This means that the system will have to be past the boot loader. For VxWorks 6.x kernels, this is not necessary. To let the tracker take advantage of this convenience and detect settings without booting the system, give the kernel version you are using to the detect command. If you are using a ROM kernel with VxWorks, you need to provide the basic VxWorks kernel binary that is later compressed to form part of the bootable image to the **vxworks-detect-settings** command. Once parameters have been detected for a certain kernel file, they can be reused using the **load-parameters** command. For more information use the built-in help for the **vxworks-detect-settings** command

The below example assumes that the system is fully booted.

```
simics> board0.software.vxworks-detect-settings param-file = vxworks.params 2
symbol-file = $boot_file
Saved parameters to vxworks.params
```

```
simics> board0.software.load-parameters vxworks.params
```

### 2.4.3 Configuration Wind River Hypervisor

The configuration of a hypervisor system involves a few more steps than for the other trackers. The first step is to find the configuration parameters associated with the hypervisor itself. Once those parameters are found, there will be on detect phase for each guest operating system that is intended to be monitored. The configuration is only meant to be run when a system is first configured. For the daily work the generated parameters file should be used. However, sometimes it can be necessary to go through the configuration steps again in order to enable new features provided by the OS awareness framework.

The **wr-hypervisor-detect-settings** command is used to generate the parameters file required to track the hypervisor itself. Remember to boot the system, at least past the boot loader, since the detect command requires the hypervisor image to be loaded into memory.

The detection of parameters for the guest operating systems, requires the hypervisor tracker to be configured and enabled. Pass the generated configuration file to the **load-parameters** command in order to configure the hypervisor tracker. Enable the tracker system with the **enable-tracker** command. To find out the node identifiers for the guest operating systems, use the **node-tree** command. For performance reason it is recommended to only configure the tracker for the guest operating systems that are of interest. The tracker will only monitor the guest operating system that have configuration parameters associated with them.

If the hypervisor system contains one or multiple Linux operating systems, the **linux-autodetect-settings** command can be used to detect the settings for them one by one. This command requires Linux to be up and running, so make sure the guest operating system is up and running. The Linux detect command also requires the operating system to be active, which is not always true under a hypervisor system. Use the **break-enter** command to stop the simulation when the Linux node is active. The **linux-autodetect-settings** command can now be used to detect the Linux parameters. There are two important things to remember when using this command on a Linux system running inside a hypervisor. The first thing is to specify the correct node identifier. The second is to specify the same parameters file as the **wr-hypervisor-detect-settings** command stored the generated parameters into, so that it can be updated with the Linux parameters as well.

If the system contains any VxWorks guest operating system that are of interest, use the **vxworks-detect-settings** command. Make sure to provide the VxWorks image that was compiled into the hypervisor. Also make sure to specify the same parameters file as before, just as in the case with Linux, this file needs to be updated with the VxWorks parameters.

The parameters file should now be ready to be loaded into the tracker system by the **load-parameters** command. Before any information about the guest operating systems are visible the simulation needs to be advanced forward. At the very least each guest operating system needs to be activated once.

This is an example of the commands required to configure a Wind River Hypervisor running two guest operating systems, Linux and VxWorks. The following configuration example requires that you have a Wind River Hypervisor running on the MPC8572 platform and access to all required trackers.



Detect the parameters for the hypervisor, by providing the hypervisor file that contains the symbols.

```
simics> run-seconds 5
simics> mpc8572ds.software.wr-hypervisor-detect-settings symbol-file = $hypervisor ?
param-file = linux-vxworks-hv.params
Saved parameters to linux-vxworks-hv.params
```

In order to detect settings for the guest operating systems the hypervisor tracker needs to be configured and enabled.

```
simics> mpc8572ds.software.load-parameters linux-vxworks-hv.params
simics> mpc8572ds.software.enable-tracker
[mpc8572ds.software.tracker info] enabling the tracker
```

Identify the node that corresponds to the Linux virtual board. It is called “linux”.

```
simics> mpc8572ds.software.node-tree
...
```

The **linux-autodetect-settings** command requires that Linux is active, so set a break point on the node that will stop the simulation once Linux is activated.

```
simics> mpc8572ds.software.break-enter -once node = linux
Added breakpoint bp53768464
```

Continue the simulation forward if it is not already running, the breakpoint will trigger as soon as Linux is activated.

```
simics> continue
Breakpoint bp53768464: mpc8572ds.soc.cpu[0] is active on node 16.
[mpc8572ds.soc.cpu[0]] v:0x001160c0 p:0x0001160c0  cmplw r5,r4
```

When detecting the Linux settings make sure to specify the same parameter file as in the case with the hypervisor settings.

```
simics> mpc8572ds.software.linux-autodetect-settings node = linux ?
param-file = linux-vxworks-hv.params
[mpc8572ds.software.tracker info] Autodetecting using 1 of 2 processor(s)
Saved autodetected parameters to linux-vxworks-hv.params
```

Detect the settings for VxWorks, using the node of the VxWorks virtual board and a file containing symbol information for the running kernel.

```
simics> mpc8572ds.software.vxworks-detect-settings node = vxworks ?
```

```
param-file = linux-vxworks-hv.params symbol-file = VxWorks
Saved parameters to linux-vxworks-hv.params
```

Finally load the updated configuration.

```
simics> mpc8572ds.software.load-parameters file = linux-vxworks-hv.params
```

### 2.4.4 Configuration OSE

The configuration of an OSE tracker can be divided into two parts. The first part is to generate a configuration file that contains the information about the loaded software, required by the OSE tracker. This step is done with the **ose-detect-settings** command. It is worth noting that detecting settings is normally only needed after a new kernel image has been created. However, it might sometimes be needed in order to enable new tracker features as well. The second step is to load the generated parameters file into the tracker system. This can be done with the **load-parameters**. After these two steps the tracker is ready to be used. In order for the **ose-detect-settings** command to succeed, the system binary has to be loaded into memory. A good time to detect the settings is usually right after the boot loader has completed, or when OSE is fully booted. However, if the OSE version is given to the detect command, the system does not need to be booted. For more information use the built-in help for the **ose-detect-settings** command.

Below follows an example of how to configure and use the OSE tracker.

```
simics> ebony.software.ose-detect-settings symbol-file = $kernel_syms 2
param-file = ose.params
Saved parameters to ose.params
```

```
simics> ebony.software.load-parameters ose.params
```

```
simics> ebony.software.node-tree
...
```

```
simics> ebony.software.list
...
```

### 2.4.5 Configuration QNX

The configuration of a QNX tracker can be divided into two parts. The first part is to create a configuration file that contains the information that the software tracker needs in order to track the system. This step is done with the **qnx-detect-settings** command. It is worth noting

that detecting settings is normally only needed after a new kernel image has been created. However, it might sometimes be needed in order to enable new tracker features. The second step is to load the generated parameters file into the tracker system. This can be done with the **load-parameters** command. After these two steps the tracker is ready to be used. For more information use the built-in help for the **qnx-detect-settings** command.

Below follows an example of how to configure and use the QNX tracker.

```
simics> ebony.software.qnx-detect-settings symbol-file = $kernel_syms 2
param-file = qnx-6.4.params
Saved parameters to qnx-6.4.params
```

```
simics> ebony.software.load-parameters qnx-6.4.params
```

```
simics> ebony.software.node-tree
...
```

```
simics> ebony.software.list
[ebony.software.tracker info] enabling the tracker
```

Process	PID
proc/boot/procnto-booke_g	1
proc/boot/devc-ser8250	2
proc/boot/slogger	3
proc/boot/pipe	4
proc/boot/ksh	5

### 2.4.6 Configuration Partition

The configuration of the partition tracker can be divided into two phases. The first phase is to generate a configuration file, normally known as parameter file. This is done by the **partition-settings** command. Each invocation of this command either adds or updates one partition with one or all processors. The second phase is the loading phase, this is when the tracker system gets configured based on the parameters file. This is done with the **load-parameters** command.

Below follows an example of how to configure the partition tracker.

```
simics> ebony.software.partition-settings partition-name = "Processors" -all 2
param-file = partition.params
Saved parameters to partition.params
```

```

simics> ebony.software.partition-settings partition-name = "Empty" -update
param-file = partition.params
Saved parameters to partition.params

simics> ebony.software.load-parameters partition.params

simics> ebony.software.node-tree
...

simics> ebony.software.list
[ebony.software.tracker info] enabling the tracker
Partitions  Processors
Processors  ['ebony.soc.cpu']
Empty      []

```

## 2.5 Software Domains

A software domain is a system or subsystem where target software runs. All software tracking mechanisms described in this chapter work individually in each software domain without affecting each other. If a configuration contains two hardware subsystems that run different software instances, there will be two separate software trackers, each one tracking the software in one of the two subsystems.

For most configurations, an **os\_awareness** component that defines the software domain will be placed in each top-level system component, in its `software` slot. For some complex configurations it is possible to define software domains differently by placing the **os\_awareness** component elsewhere in the component hierarchy, or by configuring it to use a subset of the available processors.

See the *Model Builder User's Guide* for more information about software domains.

## 2.6 Limitations

Simics OS awareness is known to work with many different OSs, including different versions. See section 2.1 for more details. The OS awareness framework relies on some data structures in order to monitor the system. That means that OS awareness might not work with a new version of a OS if it contains changes to those data structures that breaks the OS awareness logic. Many OSs supports special build configuration options that allows for turning on or off special features. Such configuration options might also affect the OS awareness if those

features changes the layout of the data structure that are monitored by the OS awareness framework.

## Chapter 3

# Debugging Target Code

The 4.6 release of the Wind River Simics simulation platform includes the ability to debug target code using a combination of Simics and the Eclipse development environment. The debugging solution for Simics relies on Eclipse CDT, the C/C++ development set of plug-ins for Eclipse enabling source level, multi-core, and multi-target debugging for Simics.

You will first need to install Eclipse 3.6 or later with the CDT plug-ins, and then install the Simics plug-ins for Eclipse. For instructions on installing Simics plug-ins for Eclipse, see the *Wind River Simics Installation Guide: Installing the Simics Eclipse Tools*.

If you're using Eclipse 3.7 (Indigo) please make sure to update the CDT plug-ins to use the latest version. This can be done by first enabling the CDT update site under **Help** → **Install New Software...** and then clicking **Help** → **Check for Updates**.

You can also use the debugger from the command line. This is described in section [3.8](#).

---

**Note:** This debugging ability is currently in beta.

---

### 3.1 Debugging Environment Tour

This section describes the debugging environment when using Eclipse with CDT. If you are already familiar with these concepts, you may decide to skip this section.

#### 3.1.1 Debug Perspective

When debugging programs Eclipse provides the user with a *Debug Perspective*, a layout of views in the Eclipse window all related to debugging.

To access the Debug Perspective, go to the menu and select **Window** → **Open perspective** → **Debug**. If Debug does not appear, choose **Other**. Eclipse will open a new window with a list of perspectives. Select the **Debug** perspective, and click **OK**.

The following figure shows the default Debug Perspective (the one you can see when opening it for the first time).

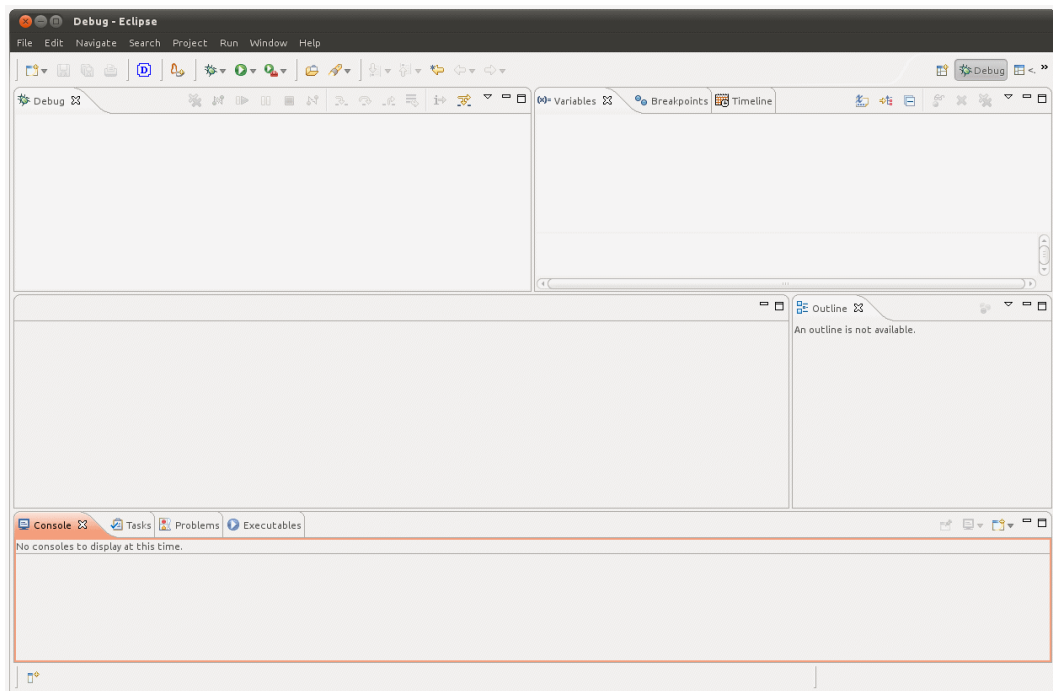


Figure 3.1: Eclipse CDT Debug Perspective.

#### 3.1.2 Views

The Debug Perspective is basically a set of views. Several views listed in the following table are specific to debugging.

##### Debug view

Lists the debug sessions and underneath the debug contexts (processes and threads) in a tree-like fashion. This is the view you will use to control the target (run, suspend, step, and so on) using the icons at the top of the view.

##### Breakpoints view

Lists the breakpoints. You can configure the breakpoints in the list, enabling or disabling them individually.

##### Registers view

Lists the registers of the target CPU, or in the case of a process, the thread.

##### Variables view

Lists the variables in the current scope and their values. This view is useful only when doing source-level debugging.

##### Expressions view

Allows you to define a number of expressions that will be evaluated during the debugging. Such expressions can be variables, access to a field in a structure, casting a structure pointer on an address, de-referencing a pointer, and so on.

**Memory view**

Allows you to display the target memory.

**Disassembly view**

Shows the disassembled code running on the target.

**3.1.3 Debug Configuration**

Debugging sessions inside Eclipse are always started from a *Debug Configuration*. The Debug Configuration, as its name implies, defines all the parameters needed to start the debugging session. It is also used to store debugging information during the session.

The debugging information is then used every time the user starts a debugging session with the configuration.

You are recommended to use a separate debug configuration for each Simics target setup you debug.

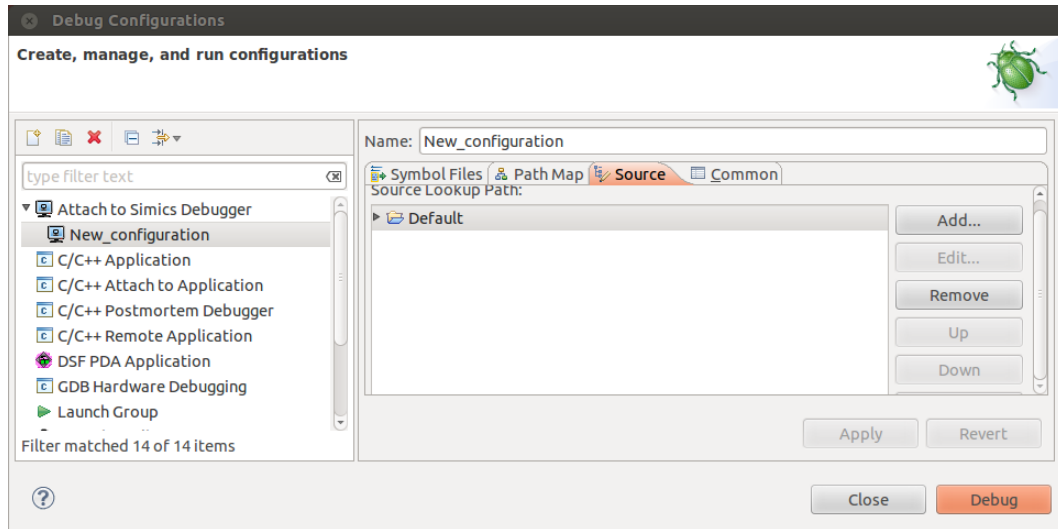


Figure 3.2: New Simics Debug Configuration.

**3.2 Getting Started with a Debug Session**

Use the following procedure to start Simics and create a debug session in Eclipse.

**3.2.1 Starting Simics**

Simics defines its sessions using a *Simics workspace*. Eclipse uses the term *workspace* for its own definitions, so in the context of Eclipse, a Simics workspace is called a *Simics Project* to distinguish it from an Eclipse workspace. When the Eclipse user interface refers to a workspace, it always means an Eclipse workspace.



You must start a Simics session before you can debug it using Eclipse. In Eclipse, select **Windows** → **Open Perspective** → **Simics**. In the Eclipse Simics Control view, click the **Launch a new Simics session** button.

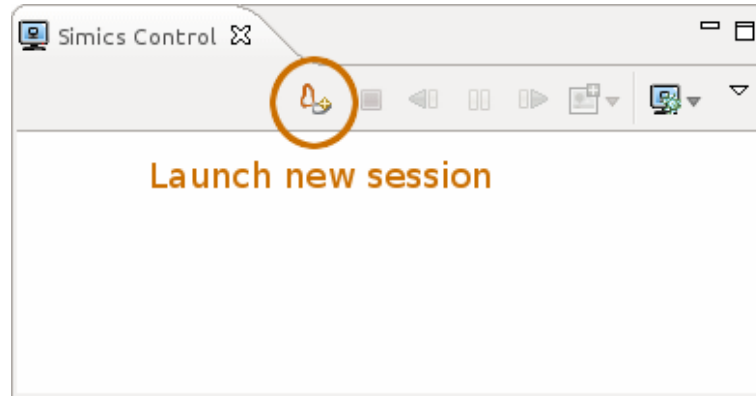


Figure 3.3: Launch Simics Session.

In the dialog that opens, specify a project name in the Project field. Select the **Script** radio button and navigate to the script you want to use, then click **OK**.

#### 3.2.2 Attaching the debugger to a Simics Session

In the Simics Control view, select a Simics session and click the **Attach the Simics TCF Debugger** button.

If you launched Simics outside of Eclipse, first execute the command **new-tcf-agent** to instantiate the tcf-agent object. This is done automatically when you start Simics from Eclipse. Then click the same button. This time it will open a window with a list of Simics sessions (*TCF peers*) running on your machine. Select your instance of Simics and click ok.

Eclipse launches the configuration and connects the debugger to your Simics session. Information on your Simics setup appears in the Debug view. You may need to open the **Debug Perspective** manually if Eclipse did not switch to it: **Window** → **Open Perspective** → **Debug**.

---

**Note:** The **Attach the Simics TCF Debugger** button displays a drop-down list of the created *Debug Configurations*. You can select **New Configuration** to create a new debug configuration. If you choose this option, the new configuration is automatically given the name of the Simics session you connect to. You can edit the configuration later by selecting **Run** → **Debug Configurations** in the Eclipse toolbar, see 3.1.3.

---

### 3.3 Setting Up Symbol Information

To enable debugging, you must provide Eclipse with symbol information. You must perform these steps separately for each simulated target that you want to debug. Symbol file infor-

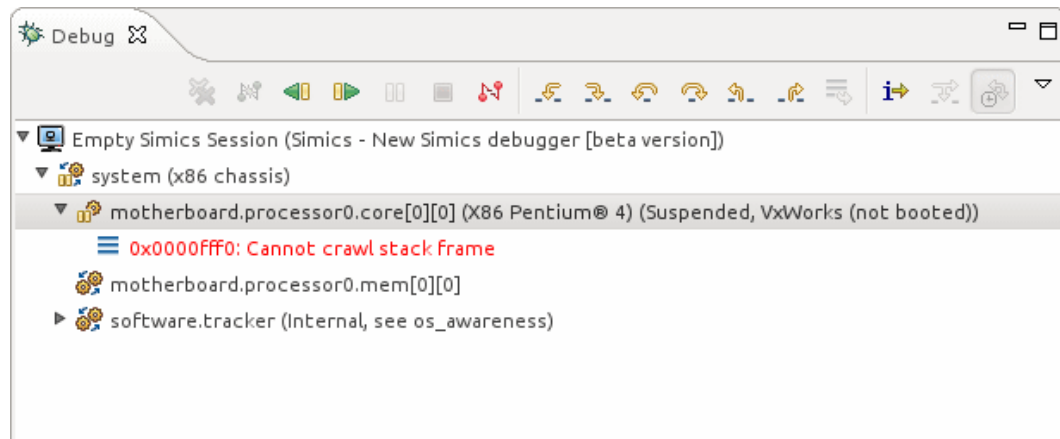


Figure 3.4: Debug View: you are now ready to debug.

mation is stored in the debug configuration file in the Eclipse workspace. It is reused by all debug sessions started by the same debug configuration.

---


**Note:** The debug symbol information must be set up for each program, kernel module, or OS kernel that you want to debug.

---

The following sections describe several cases you may encounter when setting up symbol information.

### 3.3.1 Adding a symbol file

There are two ways to add a symbol file, but eventually both ways lead to the **Symbol Browser View**. To open this view: **Window** → **Show View...** → **Symbol Browser**.

- The first way is to select the context onto which you want to add the symbol file in the **Debug View** and right-click. In the context menu, select **Symbol Files**.
- The second way to add a symbol file is to select a context in the **Debug View** and in the **Symbol Browser View** click on  **Add a new symbol file**.

In both cases a dialog box prompts you to enter a file path directly, or browse your file system and select a file. Once you have specified the desired file, you have three options (listed under the **Advanced** frame:)

1. *Use file information:* the Simics debugger uses the addresses set in the file.
2. *Re-map file at address:* the Simics debugger re-maps the segment address at the address provided by the user.
3. *Map individual sections:* the user must provide re-location addresses for some of the sections found in the file.

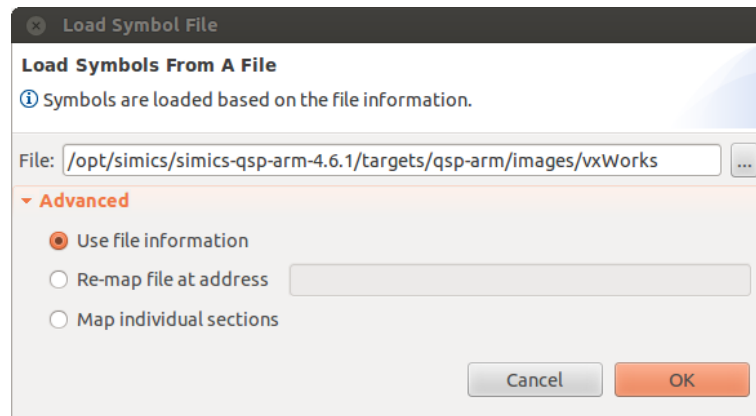




Figure 3.5: Symbol Browser View: Add new symbol file dialog.

---

**Note:** Two types of symbol file are handled by the Simics debugger: ELF *fully linked* files and ELF *relocatable* files. Depending on the file type, some options may not be available. You will see them grayed-out.

---

### 3.3.2 Re-mapping a segment or sections

Now that the symbol file is listed in the **Symbol Browser View**, you can expand its segments or sections. When the file is *fully linked*, you can re-map the segment by editing the **Address** field under the segment node. A  **marker** shows the segments that have been changed. Apply the changes by clicking on  **Apply all relocations**.

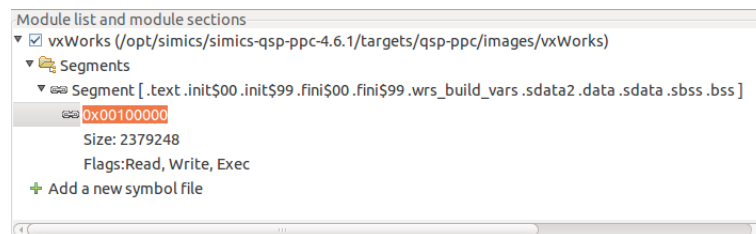



Figure 3.6: Symbol Browser View: re-map a segment.

Mapping sections is very similar. The *relocatable* sections which have not yet been mapped are marked with a  **marker**. Edit the **Address** field for the sections of interest. Again, you must apply the relocation changes.

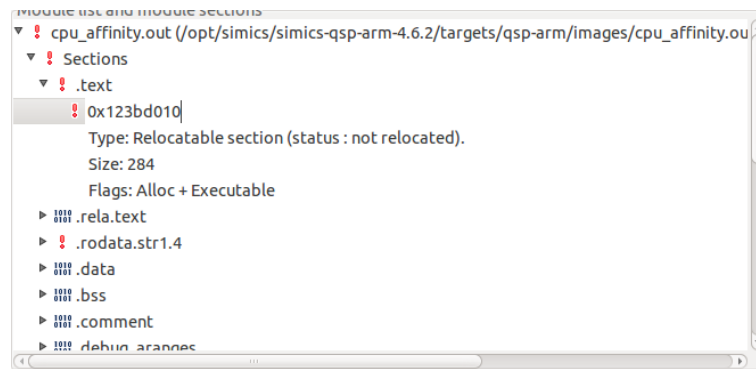


Figure 3.7: Symbol Browser View: map a section address.

#### 3.3.3 Browsing symbols

In the **Symbol Browser View**, the left-hand pane is used to add and list symbol files. When the checkbox in front of a symbol file entry is selected you can see that a list of symbols appears in the right-hand pane. Only the symbols of the selected files are displayed.




As a symbol file may list a lot of symbols, filtering is possible:

##### Filtering symbols by name

You can use the entry box to filter out the symbols by name.

##### Filtering by type

On the right, next to the entry box a set of toggle buttons are available. They are used to show or hide symbols by types. The types are:

-  Function Type
-  Data Type
-  No Type
-  Local Type

When you select a symbol in the list and right-click on it, a contextual menu appears that let the user set a breakpoint on the symbol or if more debug information are available open an editor view with the file and line where the symbol is defined.

#### 3.3.4 Symbol Browser Limitations

When using the Symbol Browser view, it is not possible to add a symbol file on a non-existing context, or to add a symbol file on multiple contexts in one step.

Both of these tasks can be performed from the Simics command line in the Simics Console view.

To add a symbol file on a non-existing context, use the following command:

```
simics > add-symbol-file symbol_file_name.elf context-query = "non_existing_context"
```

To add a symbol file on all child contexts (including cpu cores) on qsp boards, use the following command:

```
simics > add-symbol-file symbol_file_name.elf context-query = "board/*"
```

or more precisely on a specific set of cores (0 to 3:)

```
simics > foreach $i in (range 0 4) {  
..... add-symbol-file symbol_file_name.elf context-query = "\"cpu[" + $i + "]"\""  
..... }
```

#### 3.3.5 About Symbolic Debugging

Wind River's Eclipse plug-ins provide a Symbol Browser view. Use the Symbol Browser to search and filter your debug symbol file.

Unix-like systems, such as VxWorks and Linux, load ELF object files to the target. An ELF file is laid out in a number of sections, including the following:

- .bss -- uninitialized memory
- .data -- initialized memory
- .text -- executable code
- .symtab -- symbol table
- .debug -- human-readable information

The .symtab section contains the symbol table, which holds information needed to relocate a program's global and local variables, and function names (with the addresses of their entry points). All of these are called symbols.

The .debug section contains the symbols' debug information, such as source-line numbers.

Symbol information is only necessary for debugging, so it does not need to be included in the binary that is loaded on the target. The sections of the ELF file that include symbol information may be very large, so sometimes they are not included in the binary to save memory; for example, in embedded systems, where saving memory is usually a concern.

Whether the binary loaded to the target includes symbol information or not, the object file the host uses must include symbol information.

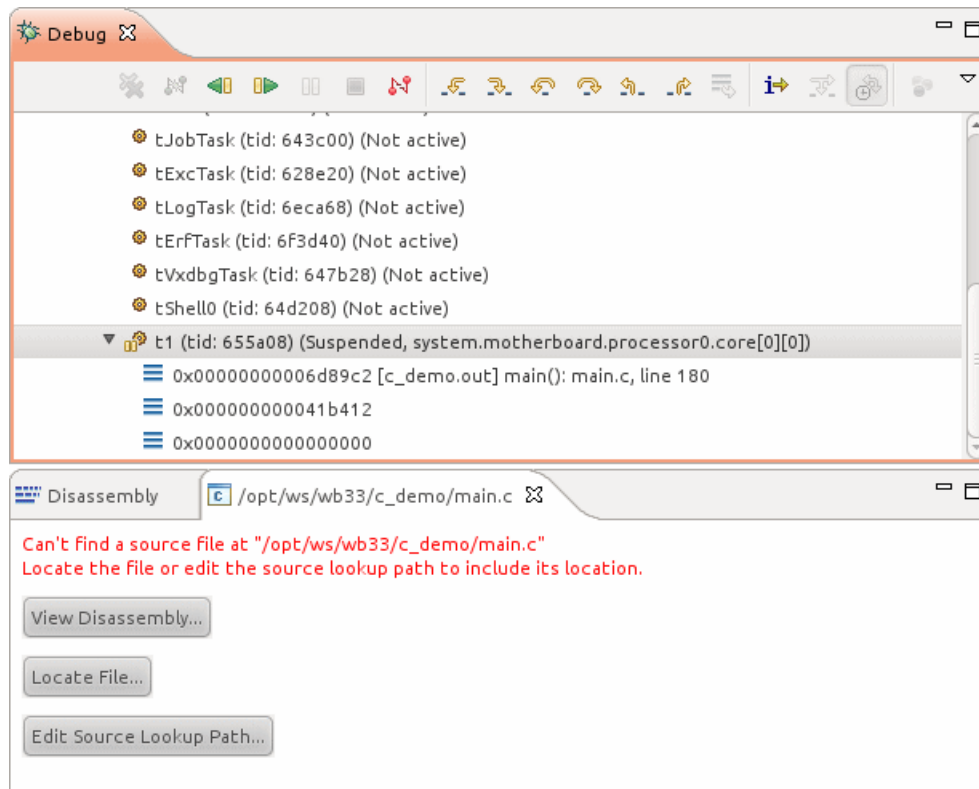
To make human-readable debugging possible, the host must use the same memory locations that the target uses for the sections of the binary. Most importantly, the host and the target must agree on the start address of the .text section.

The two areas of the Symbol Browser address both issues. The left-hand side allows you to load or relocate code to desired memory addresses for a module, and the right-hand side allows you to search and filter a module's symbol information.

## 3.4 Setting Up Source Code Path Mappings

To allow Eclipse to debug from source, you may need to point Eclipse to the source code. This is necessary when the binary file has been copied or moved from the location where it was compiled, and therefore the source code can no longer be found at the place that is compiled into the binary.

1. Find a frame that resolves to a line in the program file you want to get source for (note that this means that the program must be active in the target).



Because Eclipse does not know the location of the source file, the editor displays a “Can’t find source” error.

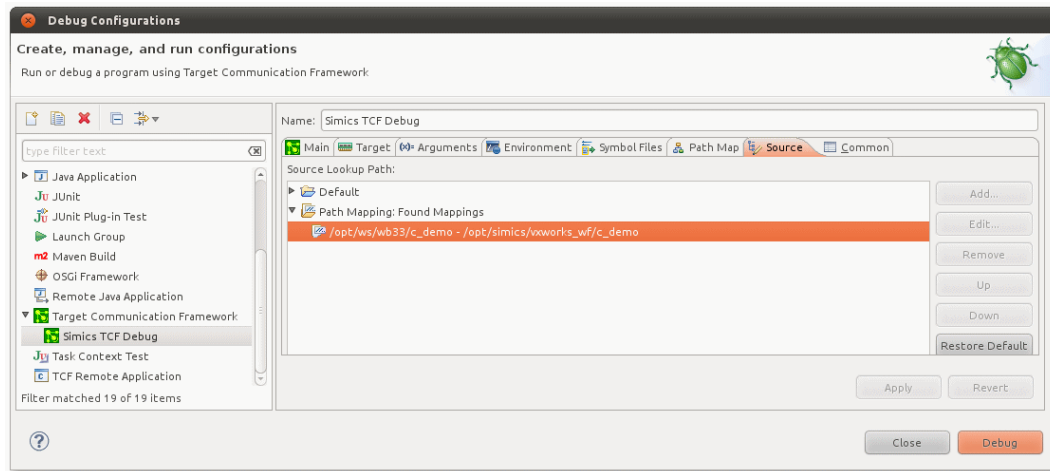
2. Click the **Locate File** button. In the dialog that appears, navigate to your source code file.

Eclipse sets the path mapping for that file. (This will also find any other program source code files located in the same host directory.) Path mappings can also be edited using the Source tab of the Debug Configuration dialog.

---

**Note:** The path mapping maps the compile-time directory path to the local file system directory. The directory can contain source files, whole projects, or multiple projects, at the user’s discretion. Clicking **Add** in the dialog box below brings up a list of supported mapping methods.

---



## 3.5 Setting Breakpoints

Use the Breakpoints view to keep track of all breakpoints, along with any conditions. You can create breakpoints in different ways: by double-clicking or right-clicking in the Editor's left overview ruler (also known as the gutter), by opening the various breakpoint dialogs from the pull-down menu in the Breakpoints view itself.

### 3.5.1 Function Breakpoints

Set a function breakpoint to stop your program when the function is called. To set a function breakpoint with an unrestricted scope (that will be hit by any process or task running on your target), select **Add Function Breakpoint (C/C++)** from the pull-down menu in the Breakpoints view. This action opens the *Function Breakpoint* dialog, where you can enter the name of the function.

### 3.5.2 Line Breakpoints

Set a line breakpoint to stop your program at a particular line of source code. To set a line breakpoint with an unrestricted scope (that will be hit by any process or task running on your target), double-click in the left gutter next to the line on which you want to set the breakpoint. A solid dot appears in the gutter, and the Breakpoints view displays the file and the line number of the breakpoint.

To adjust the properties of the breakpoint, right-click on the breakpoint in the Breakpoints view and select **Breakpoint Properties**.

### 3.5.3 Watchpoints

Set a watchpoint using any C expression that will evaluate to a memory address. This could be a function name, a function name plus a constant, a global variable, a line of assembly

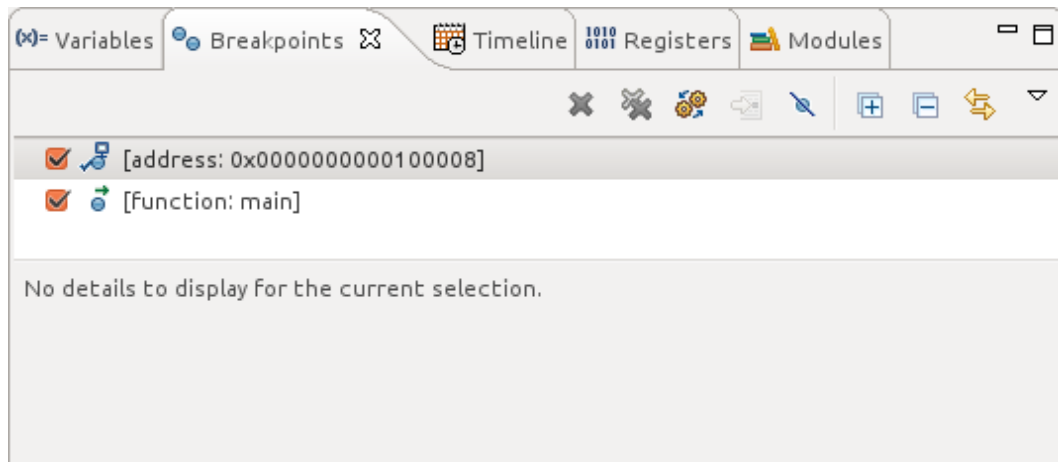


Figure 3.8: Breakpoints View: two breakpoints set.

code, or just a memory address. Watchpoints appear in the Editor's gutter only when you are connected to a task.

Conditions are evaluated after a watchpoint is triggered, in the context of the stopped task or process. Functions in the condition string are evaluated as addresses and are not executed.

Select **Add Watchpoint** from the pull-down menu in the Breakpoints view to open the *Watchpoint* dialog, where you can create and adjust the properties for the watchpoint.

### 3.5.4 Context Creation

Set a breakpoint to stop the simulation when a task or process is created.

To set a breakpoint on context creation, select **Add Context Creation Breakpoint (Simics)** from the pull-down menu in the Breakpoints view. This action opens the **Breakpoint on Context Creation** dialog, where you can enter, either the name of the task or process you want to intercept the creation, or a *Context Query*. If you leave the entry empty, the simulation will stop at every context creation.

---

**Note:** breakpoints on context creation can only be planted on contexts that have a state, in other words threads. In the case of a VxWorks RTP make sure not to use the RTP name but instead the threads name. Alternatively you may use the following context query: `rtp_name.vxe/*`.

---

### 3.5.5 Context Destruction

Set a breakpoint to stop the simulation when a task or process is exited.

To set a breakpoint on context destruction, select **Add Context Destruction Breakpoint (Simics)** from the pull-down menu in the Breakpoints view. This action opens the **Breakpoint on Context Destruction** dialog, where you can enter, either the name of the task or process



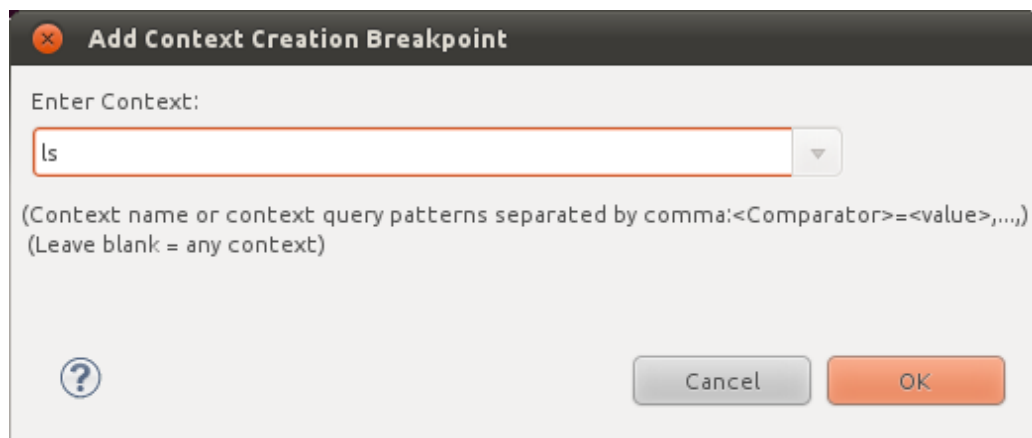


Figure 3.9: Breakpoint on Context Creation dialog.

you want to intercept the destruction, or a *Context Query*. If you leave the entry empty, the simulation will stop at every context destruction.

---

**Note:** breakpoints on context destruction can only be planted on contexts that have a state, in other words threads. In the case of a VxWorks *RTP* make sure not to use the *RTP* name but instead the threads name. Alternatively you may use the following context query: `rtp_name.vxe/*`.

---

### 3.5.6 On Physical Address

Set a breakpoint on a physical address.

By default the Simics debugger plants breakpoints using the virtual address but it is possible to force the debugger to see the address as a physical address. To set a breakpoint on a physical address, select the breakpoint in the **Breakpoints View**, right-click and select **Breakpoint Properties**. The **Breakpoint Properties Window** appears, select the **Simics Options** tab and tick the box *Physical Address*.

### 3.5.7 Ignore Count

You can specify that a breakpoint effectively breaks the simulation only after being hit a specific number of times. To add an **ignore count**, open the Breakpoints view, select a breakpoint and right-click. A contextual menu is displayed, select **Breakpoint Properties**. In the new window you can enter a **ignore count** value in the corresponding entry box.

---

**Note:** When using reverse execution, the hit count of a breakpoint is increased just like in forward execution. If you set up a breakpoint to use ignore count, it may behave differently to what you expect.

---

### 3.5.8 Manipulating Breakpoints

#### Importing Breakpoints

To import breakpoint properties from a file:

1. Right-click in the Breakpoints view and select **Import Breakpoints**. The Import Breakpoints dialog appears.
2. Select the breakpoint file you want to import, then click **Next**. The Select Breakpoints dialog appears.
3. Select one or more breakpoints to import, then click **Finish**. The breakpoint information appears in the Breakpoints view, and the next time the context for that breakpoint is active in the Debug view, the breakpoint will be planted.

#### Exporting Breakpoints

To export breakpoint properties to a file:

1. Right-click in the Breakpoints view and select **Export Breakpoints**. The Export Breakpoints dialog appears.
2. Select the breakpoint whose properties you want to export, and type in a file name for the exported file. Click **Finish**.

#### Refreshing Breakpoints

Right-click a breakpoint in the Breakpoints view and select **Refresh Breakpoint** to cause the breakpoint to be removed and reinserted on the target.

This is useful if something has changed on the target (for example, you downloaded a new module) and the breakpoint is not automatically updated.

To refresh all breakpoints in this way, select **Refresh All Breakpoints** from the pull-down menu in the Breakpoints view.

#### Disabling Breakpoints

To disable a breakpoint, clear its check box in the Breakpoints view. This retains all breakpoint properties, but ensures that it will not stop the running process. To re-enable the breakpoint, select the box again.

#### Removing Breakpoints

To remove a breakpoint:

- Right-click on a breakpoint in the Editor gutter and select **Toggle Breakpoint**.
- Select a breakpoint in the Breakpoints view and select **Remove**.

### 3.5.9 Status

To see the status of a breakpoint, select the breakpoint in the **Breakpoints View**, right-click and select **Breakpoint Properties**. The **Breakpoint Properties Window** appears, select the **Status** tab. The right pane shows the context tree and underneath each context node, whether the breakpoint is planted or not.

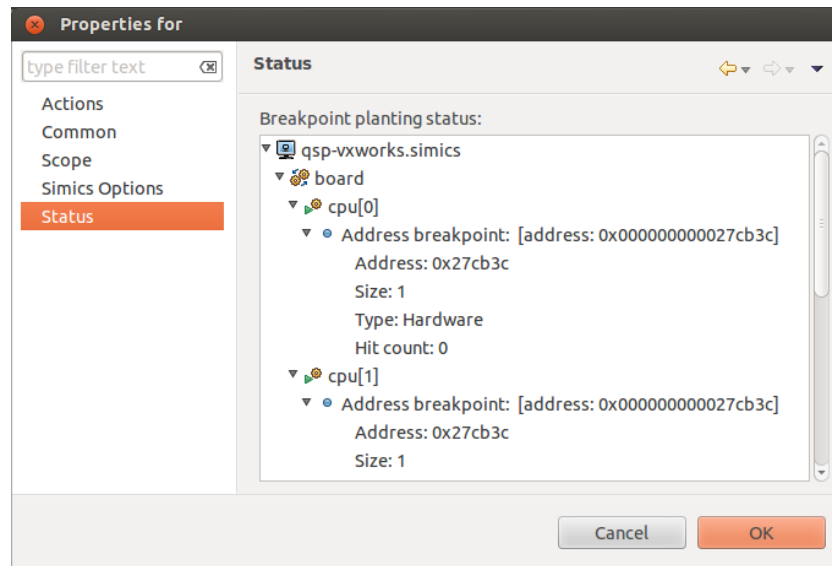


Figure 3.10: Status of a breakpoint.

### 3.5.10 Scoping

By default, when you create a breakpoint using the Simics debugger the scope of the breakpoint is set to all the existing contexts. You can change the scope of the breakpoint by right-clicking on the breakpoint in the **Breakpoints View** and then selecting **Breakpoint Properties**. The **Breakpoint Properties Window** appears, click on the **Scope** tab.

Two ways of restricting the scope of a breakpoint are offered. The first one, named **Basic** is based on the contexts tree: you can cherry-pick contexts on which you want the breakpoint to be planted. Note that selecting a parent context will automatically select all the children contexts.

The second way to scope a breakpoint, named **Advanced** is done using a *Context Query*.

### 3.5.11 Breakpoint Icons

The following icons appear on breakpoints in the Breakpoints view:

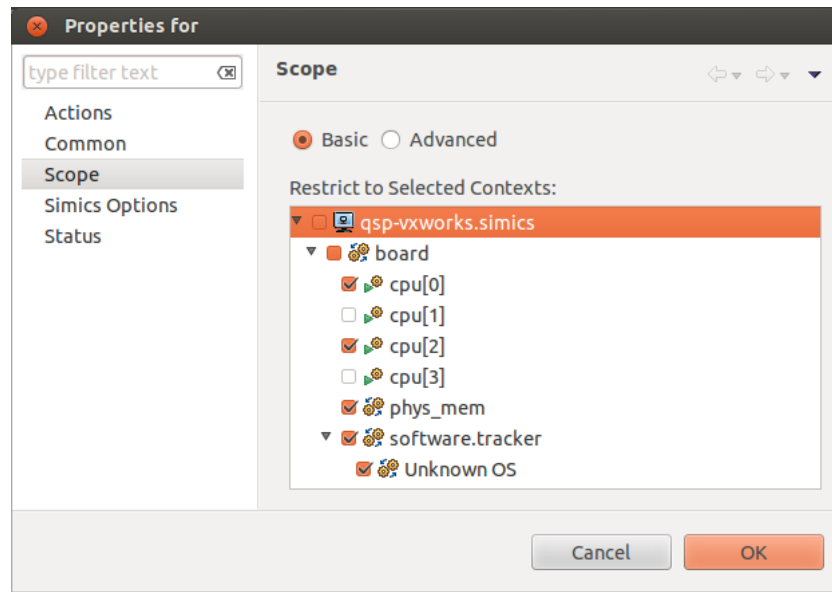


Figure 3.11: Changing a breakpoint scope using the contexts tree.

Icon	Description
	Breakpoint is disabled.
	Breakpoint is enabled and sent to the target.
	Watchpoint write access.
	Watchpoint read access.
	Watchpoint read/write access.

## 3.6 Using Context Queries

Simics implements a query-based system to specify subsets of contexts. The queries specify context properties, and what values they must have in order to match. In addition, a query can filter based on a context's ancestors in the context hierarchy.

### 3.6.1 About Context Queries

A *query* consists of a sequence of parts separated by /. This sequence specifies a path through the context tree.

A context matches the query if the last part of the query matches the properties of the context, and the parent of the context matches the query excluding the last part. The properties of a context matches a part if each property specified in the part matches the property of the same name in the context or if the name of the context matches the string specified in the part.

The contexts are assumed to be placed in a tree. Each context has zero or one parent. If it has zero parents it is a child of the root of the tree.

There are also two wild cards. The part `*` matches any context. The part `**` matches any sequence of contexts. If the query starts with a `/`, the first part of the query must match a child of the root of the context tree.

### 3.6.2 Context Query Syntax

```
query = [ "/" ], { part, "/" }, part ;
part = string | "*" | "**" | properties ;
properties = property, { ",", property } ;
property = string, "=", value ;
value = string | number | boolean ;
string = quoted string | symbol ;
quoted string = "'", {any-character - ('"' | '\')}
                | ('\ ', ('"' | '\'))}, "' ;
symbol = letter, { letter | digit } ;
number = digit, { digit } ;
boolean = "true" | "false" ;
letter = ? A-Z, a-z or _ ? ;
digit = ? 0-9 ? ;
any-character = ? any character ? ;
```

### 3.6.3 Examples

This section illustrates the syntax with some examples, and what a user might mean when providing such a query:

- **httpd**: matches all contexts named “httpd”.
- **pid=4711**: matches any context with a property **pid**, which has the value 4711.
- **/server/\*\***: matches all contexts that are descendants of the top level context named “server”.
- **“Linux 2.6.14”/Kernel/ \***: matches all kernel processes in operating systems named “Linux 2.6.14”.
- **pid=4711/ \***: matches all threads in processes with the pid 4711.
- **/server/\*\* /HasState=true**: matches all threads that are descendants of the context “server”.

### 3.6.4 Difference in the Command Line

You will notice some difference between the syntax of context queries issued in Eclipse and context queries passed as Simics command parameters. As mentioned at the beginning of this documentation, the Simics debugger is built on top of the *TCF* technology which introduces the context query concept that is very similar in principle to the process trackers node path concept.

Unfortunately both concept syntaxes differ slightly and out of compatibility concerns we modified the syntax of context queries in the command line to resemble the node path syntax. The difference stays minor as it impacts only the double quote character (") used to quote a string with non alpha-numeric characters. In the command line you must either escape this character or use the simple quote character (') instead.

Originally the context query to match a context named *rule30.elf* is written `name="rule30.elf"`. The corresponding node path is `name='rule30.elf'`. So for instance in the command line you will use:

```
simics> add-symbol-file context-query = "name='rule30.elf'"
or
simics> add-symbol-file context-query = "name=\"rule30.elf\""
```

## 3.7 Expressions

An *expression* is any statement of values (constants, variables, operators, and so on) which, when evaluated, returns an explicit value. Expressions can contain function calls.

Add expressions that interest you to the Expressions view, where you can see their values update when your program is suspended. Changed values appear highlighted in yellow; errors appear in red. Only values that were visible in the Expressions view when they changed are highlighted. You can manually change the value of some variables and registers, but some things (such as read-only memory) cannot be changed. By default, the values in the Expressions view reflect whatever debug context is currently selected in the Debug view. If a symbol is out of scope, no value appears.

### 3.7.1 Populating the Expressions View

When you add an expression, the expression and its value appear in the Expressions view. When the execution of a program is suspended, all expressions are reevaluated.

You can add items to the Expressions view in several different ways.

- Type the symbol, register (e.g. *\$r1*), variable name, or expression directly into the **Expression** column, just as you would add information to a spread sheet.

---

**Note:** If the Expressions view contains an assignment, or a function call with a side effect, or anything else that changes the target state, it is executed on each suspend. So, for example, if you type `i=11`, `i` is reset to 11 each time the program suspends.

---

- Right-click in the Expressions view, select **Add Watch Expression**, and type the expression that you want to evaluate. For example,  $(x-5)*3$ .
- Drag and drop a particular register, variable, or expression from another view into the **Expression** column.
- Right-click a variable in the Variables or Registers view and select **Watch**.
- Highlight a symbol in the Editor, then right-click in the Editor and select **Watch *symbol\_name***.

### 3.7.2 Adding Expressions

You can monitor:

#### Value at an address

You can monitor the value at an address by using expressions. In the **Registers View** enter the address and cast it as a pointer to the desired type. For instance `*((char *)0x12345678)` or `(struct my_struct *)0x116F0088`.

#### Registers

To add a register to the **Expressions View**, right-click on the register and select **Watch In Expression**. You can see that the register appears in the **Name** column using the syntax: `$register_name`. You can of course use this syntax directly to add a register to the **Expressions View**.

### 3.7.3 Deleting Expressions

Right-click one or more items in the Expressions view and select **Remove** to delete them.

- To clear the view, select **Remove All**.
- You can also use the **Remove** and **Remove All** buttons in the Expressions view toolbar.

## 3.8 The Command Line Debugger

In addition to the Eclipse frontend the Simics debugger has a command line frontend. You can use it either stand-alone or together with the Eclipse frontend.

This section provides an overview of the command line frontend to the debugger and how its pieces work together. The tutorial in *Getting Started With Simics* describes how to use these commands in a complete workflow. For details about the commands see the *Wind River Simics Reference Manual*.

To step through your code, inspect the value of variables, and so on, you use debug contexts. Each debug context mirrors one part of the target system. It can be a processor, the memory space of a processor or a software abstraction like a process or thread. There are also debug contexts providing grouping for other contexts, but nothing else. Section [3.8.1](#) describes debug contexts in more detail.

The debugger will only create debug contexts for the software in the system if you have configured OS awareness and enabled it. How to set up OS awareness is described in chapter 2. If you do not use OS awareness you can debug directly on the processors instead, but then the debugger will not track when the operating system on the target switches between different processes or tasks.

To perform source level debugging you need to tell the debugger about the binaries, i.e. executable files and libraries, the part of the system you want to debug is running. This is described in section 3.8.2. That section also tells you how to tell the debugger where to find the source code for your program if needed.

How manage break- and watch-points with the command line frontend is described in section 3.8.3.

---

**Note:** This section only describes the C/C++ debugger specific concepts. You can still use any other Simics commands to help you debug your code.

---

### 3.8.1 Debug Contexts

The debugger presents the target system as a set of debug contexts. Each debug context represents one interesting part of the system. It can be a software concept, like a process, a thread or a task, or a processor core or the physical memory space of a processor. To provide context there are also debug contexts which provide grouping, like a machine, or a group of all user space programs on a Linux system.

When you interact with debug contexts directly you usually interact with a context running code, like a thread or a processor core. In this text we call such a debug context a thread. A thread allows you to step through the code and to inspect variables scoped to the current location in the code.

By default Simics does not enable access to the new debugger from the command line. Global stepping commands, etc will be directed at the current processor in Simics instead. There are two ways to enable the debugger and get access to debug contexts. The first is to configure the debugger with symbol or path information or setting a debugger breakpoint as described in the following sections. This will make Simics track debug contexts. The second way is to explicitly select a debug context with the **debug-context** command. This will set a current debug context and will make Simics track debug contexts.

Once Simics tracks debug contexts, Simics will update the current debug context to the currently running thread every time the simulation stops. The current debug context is the debug context all global step commands and inspections commands will interact with. This means that when you have hit a breakpoint or completed a step you can use the global commands to investigate the state of your software and to continue stepping through the code.

Once you have a debug context you can use it to step through the code and inspect its state. Here is a summary of the commands you can use:

#### **step-line, reverse-step-line**

Run forward or in reverse until the debug context reaches another line in the program.

This is the same as the Step Into and Reverse Step Into buttons in Eclipse.



**next-line, reverse-next-line**

Run forward or in reverse until the debug context reaches another line, but skip over calls made by the current function. This is the same as the Step Over and Reverse Step Over buttons in Eclipse.

**finish-function, uncall-function**

Run forward or in reverse until the debug context reaches a line and the top-most stack-frame is no longer on the stack. This is the same as the Step Return and Uncall buttons in Eclipse.

**step-instruction, reverse-step-instruction**

Run forward or in reverse until the debug context reaches another instruction. This is the same as the Step Into and Reverse Step Into buttons in Eclipse when using Instruction Stepping mode.

**next-instruction, reverse-next-instruction**

Run forward or in reverse until the debug context reaches another instruction, but skip over calls made by the current function. This is the same as the Step Over and Reverse Step Over buttons in Eclipse when using Instruction Stepping mode.

**sym-value**

Get the value of a symbol or a C expression.

**sym-type**

Get the type of a C expression.

**sym-address**

Get the address of a C lvalue expression.

**sym-string**

Get the contents of a C string identified by a C expression.

**list**

List source code.

**sym-source, sym-file**

Show where an address or function can be found in the source code.

An active thread also have a stack of call frames, or just frames. Simics provides commands to show the stack and to select which frame other commands should use to find local variables. The stack goes from the innermost frame up towards the outermost frame. The current stack frame is reset to the innermost frame every time Simics stops executing.

**stack-trace**

Show the stack.

**frame**

Select the frame with the given number.

**up**

Select a frame outside the currently selected stack frame.

**down**

Select a frame inside the currently selected stack frame.

### 3.8.2 Configuration

To perform source level debugging you need to tell the debugger about the binaries the debug contexts you want to debug are running. These binaries must be in ELF format and should contain debug information in Dwarf format. Once this is done you can set breakpoints, step through your program and inspect the value of variables in it.

---

**Note:** If you try to debug optimized code you may not be able to view all variables and the mapping between line information and addresses in memory may be confusing.

---

You configure the binaries using the **add-symbol-file** command. The command uses *context queries* to limit which debug contexts the information applies to. You can read more about context queries in section 3.6. Here is a summary of the commands to manage the memory map:

**add-symbol-file**

Add a symbol file or a section or segment from a symbol file to the memory map.

**clear-memorymap**

Clear all memory map entries set from the Simics command line.

**show-memorymap**

Show all memory map entries set from the Simics command line.

**list-segments, list-sections**

List segments or sections in an ELF file.

For simple programs you just have to provide the main binary, but for more complex cases, with shared libraries or dynamically loaded modules you may need to add several symbol files for your program.

If the source code paths in the debug information in the binaries do not match the location of the source code on your host system you also need to tell the debugger how to find the source code if you want the debugger to be able to show you the actual source code.

You tell the debugger where to find the source code for your program using the **add-pathmap-entry** command. As with the **add-symbol-file** you can use context queries to limit which debug contexts the information applies to. Here are the commands you can use to manage the path translation:

**add-pathmap-entry**

Add a translation from a path in the debug information to a path on the host.

**clear-pathmap**

Remove all translations of paths set from the command line.

**show-pathmap**

Show all translations of paths set from the command line.

**3.8.3 Breakpoints**

The C/C++ Debugger allows you to set breakpoints and watchpoints on source code lines and on variables and expressions. These breakpoints also show up in Eclipse. Breakpoints and watchpoints are set with two simple commands:

**break-line**

Add a breakpoint or watchpoint on a source line.

**break-location**

Add a breakpoint or watchpoint on a location given as an address or as a C expression.

Both commands take flags to say if they should trigger for *reads*, *writes* and/or *execution*. A watchpoint is simply a breakpoint with the *read* or *write* flag set. In this section we use the term breakpoint to mean both watchpoints and breakpoints.

Breakpoints can be set on all contexts matching a context query. This allows you to set breakpoints which will trigger in threads or processes that do not exist yet. The default context query matches all debug contexts.

You can also set breakpoints for a specific debug context by using the **break-location** and **break-line** commands that are namespaced on debug contexts. This allows you to limit the breakpoint to a particular debug context, but this can only be done for debug contexts that already exists.

Breakpoints set with **break-line** and **break-location** are managed by commands namespaced on the **breakpoints** object in Simics. Here are the commands you can use to manage such breakpoints:

**breakpoints.list**

List breakpoints.

**breakpoints.show**

Show details about a breakpoint.

**breakpoints.enabled**

Set or get if the breakpoint should trigger.

**breakpoints.ignore-count**

Set or get the ignore count of a breakpoint.

**breakpoints.temporary**

Set or get if a breakpoint should be removed after it has triggered.

**breakpoints.delete**

Delete a breakpoint.

---

**Note:** The **breakpoints** object only manages breakpoints set with **break-line** and **break-location**. Other kinds of breakpoints in Simics are managed with separate commands.

---

## 3.9 Debugger Limitations

This section lists limitations for the Simics debugger in the current release of Wind River Simics.

### General

- The Simics debugger only supports x86 and x86\_64, PPC32 and ARM targets.
- When using the Symbol Browser view, it is not possible to add a symbol file on a non-existing context, or to add a symbol file on multiple contexts in one step. Both of these tasks can be performed from the Simics command line in the Simics Console view. from the command line. See [3.3.4](#).
- On all targets, the stack trace will work only if a symbol file has been loaded.
- The Simics debugger implements very basic C++ support. Only simple C++ expressions are supported in the expression evaluator.
- The Simics debugger GUI does not currently support scripting.
- When using the skip-to command, the UI is not refreshed if you skip to a bookmark that is after the current time of the simulation.
- It is not possible to reboot an OS if Simics OS awareness trackers are enabled.
- You can not disassemble memory using a physical address.

### Process trackers

- Process trackers have been tested on VxWorks, Linux, and Wind River Hypervisor. OSE and QNX are supported by Simics OS awareness, but they have not been tested with the Simics debugger.
- When using VxWorks as a guest OS with a Hypervisor system, it is not currently possible to debug a real-time process (RTP) on the VxWorks virtual board.
- When debugging SMP systems with no process trackers configured, the debugger can not handle thread migration over cores. Make sure you added symbol files and planted breakpoints on all the cores.

### Registers

- Simics processor models do not currently provide full register details. This is a limitation of the models, not a limitation of the debugger.
- When a processor has registers overlapping, changing one register does not refresh automatically the other register value. You need to manually click the **Refresh icon** in the **Registers View**.

### Breakpoints

- Planting breakpoints on *Processes* or *RTPs* contexts does not work. Instead the user-interface plants breakpoints on all the existing children contexts. In the case a new thread is created in this *process*, the breakpoint will not be automatically planted on it.
- When editing multiple source code files with the same name, line breakpoints are planted on all files regardless of the full path of the files. You can scope more precisely the breakpoint in the Breakpoint Properties window, under the tab Scope.
- You cannot plant a breakpoint directly from the Disassembly view in the editor's panel (the one that is opened when clicking on a context in the debug view). This is a bug in the 8.0 version of the Eclipse CDT. The issue does not appear with CDT 7.0.

The workaround is to manually open a Disassembly view in the viewers panel by selecting **Window** → **Show View** → **Disassembly**. In the instance of the view that opens, you can plant breakpoints by double-clicking on the address.

- When reverse stepping over a range, breakpoints in this range will be skipped. However breakpoints outside of the range (for example in a function called inside the range) will be hit as usual.
- You can not plant a breakpoint on a local variable.

# Chapter 4

## Debugging

The most obvious reason to track individual processes in the target system is to be able to debug them. The *Hindsight User's Guide* describes how to use Simics as a debugger, or together with an external debugger. The addition of software tracking enables the debugger to easily follow just one process.

### 4.1 Contexts

Since Simics is a full-system simulator, every instruction executed and every memory transaction performed is equally visible. This is undesirable if you are only interested in what happens within one part of the software, such as a single process or task. Simics uses *context objects* to represent processes running on the target, allowing you to set breakpoints in the virtual address space of a single process.

See the *Hindsight User's Guide* for more information about context objects.

### 4.2 Symbol Tables

To be able to do source-level debugging of the software, information about the symbols in the binary code and debug information can be loaded into Simics. This is handled by **symtable** objects.

The **track** command automatically creates a symtable object and attaches it to the context, but doesn't load any information in it. This can be done with commands such as **<symtable>.load-symbols**:

```
simics> $ctx->symtable.load-symbols myprog
```

In some cases, it is simpler to create the symtable object beforehand and pass it to the track command, which will then use that symtable instead of creating a new one:

```
simics> new-symtable -n mysyms myprog
simics> mpc8641d_simple.software.track myprog symtable=mysyms
```

This allows it to be reused for several tracked processes, for instance.  
See the *Hindsight User's Guide* for more information about symbol tables.

## 4.3 Using Simics Contexts with GDB

Simics supports using external debuggers, such as GDB, talking to modules inside Simics. The **new-gdb-remote** command, as described in the *Hindsight User's Guide* enables a GDB service that lets GDB connect to Simics and debug the software running on the simulator. To use GDB to debug a single process, you can create a tracker and then pass the context object as the *context* parameter to the **new-gdb-remote** command. This will make the remote GDB session debug only this context.

```
running> mpc8641d_simple.software.track myprog  
Context myprog0 will start tracking myprog when it starts  
running> new-gdb-remote context = myprog0  
[gdb0 info] Awaiting GDB connections on port 9123.  
[gdb0 info] Connect from GDB using: "target remote localhost:9123"  
[gdb0 info] Attached to myprog0
```

## Chapter 5

# Code Coverage

Using Simics, it is possible to do coverage analysis of the code running on the target system. This analysis is done using the unmodified binary running on the simulated system by profiling which machine instructions are executed. This profile is then mapped to the source code using debug information generated by the compiler and presented so that it is possible to see which source lines have been executed.

### 5.1 Requirements

To be able to perform code coverage analysis, Simics needs to support instruction profiling. Instruction profiling is currently only available for SPARC-V9, PowerPC, x86, ARM and MIPS targets. Furthermore, it requires that Simics is started in *stall mode*. See the *Hindsight User's Guide* for more information about simulation modes.

Furthermore, the software tracker needs to be loaded with a configuration matching the software running on the simulated system. See section [2.2](#) for more information.

### 5.2 Limitations

Simics can do precise profiling of the execution of all instruction in the program running on the target system, but it should be noted that there are some things that are needed to produce results that are as useful as possible.

The command described in this chapter will only profile the code running inside a single program, and not the code running in other programs cooperating with the program under analysis. Neither will it produce coverage analysis of code executed in the operating system kernel on behalf of the program. Simics could certainly do these things, and it is possible to do it using the more advanced techniques described later in this document. But this chapter focuses on the more common case of profiling a single program.

The source-level information relies on Simics being able to map instruction addresses to source lines. There are at least two separate obstacles that can prevent Simics from doing that.



First of all, to be able to do coverage of relocated code, such as when using a dynamically loaded library, the user must manually tell Simics at what address the library code is loaded, since it can not be detected automatically.

Secondly, it relies on useful debug information to map individual instructions to source lines. This is often problematic if the binary has been compiled with optimization. Some compilers refuse to emit debug information for optimized code, and some compiler emit it, but with results that can be confusing or hard to interpret. So to get the best result from the coverage analysis, it may be necessary to recompile the binary with less optimization.

Finally, to get all the required information from the binary, it needs to be available to Simics as a file that is identical to what is running on the target system.

## 5.3 The code-coverage Command

The **code-coverage** command works in a similar way to the **track** command described in section 2.3.

Example binary and source code is provided by the Firststeps package and located in the target directory. Copy the source code into the host's `/tmp` directory; use Simics FS to copy the binary into the target system.

```
simics> mpc8641d_simple.software.code-coverage node = "name='worker-opt'" 2
binary = /tmp/worker/worker-opt output = worker-opt-coverage1
```

The above tells Simics to look for a file called `worker-opt` and load debug information from it. Simics will then watch for new processes called `worker-opt` starting on the target system. Immediately when the process is started, the profiling begins and continues until the process exits. When the process exits, the collected low-level profiling data is processed and the output is generated and saved to the directory name you gave the `output` parameter.

Start the `worker-opt` program on the target.

```
~ # worker-opt 1 2
```

The generated output is a bundle of HTML files with an `index.html` file as the entry point, see figure 5.1. It contains both source-level, see figure 5.2, and assembly-level coverage information, see figure 5.3.

Another common way of running the **code-coverage** command is by giving it a **symtable** object instead of the file name of a binary. The symtable object should be loaded with the debug information needed. In fact, giving the binary file is just a shortcut to the simple case of creating a symtable and loading the information from the binary with the default parameters. This sequence of commands is equivalent to the one given above.

```
simics> new-symtable syms /tmp/worker/worker-opt
simics> mpc8641d_simple.software.code-coverage node = "name='worker-opt'" 2
symtable=syms output=worker-opt-coverage2
```

Coverage analysis		
Source Files   <a href="#">Disassembly</a>		
/home/jakob/Development/cross-ppc750/crosstool-0.27/build/powerpc-750-linux-gnu/gcc-3.3.2-glibc-2.3.2/build-glibc/csu/crti.S	21 / 23	91.3 %
/home/jakob/Development/cross-ppc750/crosstool-0.27/build/powerpc-750-linux-gnu/gcc-3.3.2-glibc-2.3.2/build-glibc/csu/crtn.S	8 / 8	100.0 %
<a href="#">/tmp/worker_opt.c</a>	33 / 43	76.7 %

Figure 5.1: Code Coverage HTML Overview

Using an explicit symtable is much more flexible. It can be loaded with information from several files, when the running binary is using code loaded from different libraries. It also allows specifying the address where the code is loaded. See the *Hindsight User's Guide* for more information about symbolic debugging and symbol table objects.

---

**Note:** *Why does my source coverage look so strange?* Remember that when running optimized binaries, the executed instructions do not always map to exactly one line, and the compiler often produces less useful debug information.

---

It is also possible to generate a simple text file with all the source-level coverage information for further processing with other tools. Use the `format=raw` flag to the **code-coverage** command to switch to this output format.

See the documentation of the `<os_awareness>.code-coverage` command for details.

## 5.4 Saving Manually

The profiling is done by a **coverage\_profiler** object that is created and returned by the **code-coverage** command. When an output location is given to the command, the profiler object will save the profiler and destroy itself when the process finishes.

However, if no output location is given to the **code-coverage** command, no output will be created automatically when the profiled process finishes. Instead, the `<coverage_profiler>.save` can be used to save the generated coverage profile. It takes the same *format* and *output* parameters as the **code-coverage** command. It is possible to run the **save** command at any time, even before the process is finished.

```
simics> mpc8641d_simple.software.code-coverage node = "name='worker-opt'"
binary = /tmp/worker/worker-opt
simics> continue
simics> coverage_nameworkerop_2.save coverage
```

## Coverage analysis

[Source Files](#) | [Disassembly](#) | /tmp/worker\_opt.c

Covered 33 of a possible 43 lines (76.7%).

```

1  /* Example program used to demonstrate the timeline view and code coverage */
2
3  /*
4   * This Software is part of Wind River Simics. The rights to copy, distribute,
5   * modify, or otherwise make use of this Software may be licensed only
6   * pursuant to the terms of an applicable Wind River license agreement.
7   *
8   * Copyright 2012 Intel Corporation
9   */
10
11 #include <stdio.h>
12 #include <stdlib.h>
13 #include <stdint.h>
14 #include <pthread.h>
15 #include <unistd.h>
16
17 static int work_done = 0; /* How much work has been processed */
18 static pthread_mutex_t work_mutex = PTHREAD_MUTEX_INITIALIZER;
19 static pthread_cond_t work_cond = PTHREAD_COND_INITIALIZER;
20
21 static int fib(int f) {
22     if (f < 2) {
23         return f;
24     }
25     if (f == 2) {
26         return 1;
27     }
28     return fib(f - 2) * 2 + fib(f - 3);
29 }
30
31 static void
32 report_work_done(void) {
33     work_done++;
34     if(pthread_cond_signal(&work_cond)) {
35         fprintf(stdout, "pthread_cond_signal error\n");

```

Figure 5.2: Code Coverage Source Coverage

## Coverage analysis

[Source Files](#) | Disassembly

**/tmp/worker-opt**

**section .init**

**\_init**

```
10000440: 94 21 ff e0 stwu r1,-32(r1)
10000444: 7c 08 02 a6 mflr r0
10000448: 90 01 00 24 stw r0,36(r1)
1000044c: 48 00 00 41 bl 1000048c
10000450: 48 00 01 11 bl 10000560
10000454: 48 00 08 89 bl 10000cdc <__do_global_ctors_aux>
10000458: 80 01 00 24 lwz r0,36(r1)
1000045c: 7c 08 03 a6 mtlr r0
10000460: 38 21 00 20 addi r1,r1,32
10000464: 4e 80 00 20 blr
```

Figure 5.3: Code Coverage Assembly Coverage

## Chapter 6

# Profiling Tools

Unlike most profiling tools, which instrument the target source code or object code, Simics can profile a workload non-intrusively. This allows you to profile without disturbing the execution. Simics will profile arbitrary code, including device drivers, dynamically generated code, and code for which you do not have the source. Also unlike most profiling tools, Simics collects profiling data exactly, not by sampling the execution and relying on statistics.

The following sections describe how to make Simics collect profiles of instruction execution and memory reads and writes, and how to examine the collected data.

### 6.1 Instruction Profiling

---

**Note:** Instruction profiling is currently only available for SPARC-V9, PowerPC, x86, ARM and MIPS targets. Furthermore, it requires that Simics is started in *stall mode*. See the *Hindsight User's Guide* for more information about simulation modes.

---

Simics can maintain an exact execution profile: every single taken branch is counted, showing you exactly what code was executed and how often branches were taken.

To get started with instruction profiling, type **mpc8641d\_simple.soc.cpu[0].start-instruction-profiling** at the prompt (assuming the machine you are simulating has a processor called **mpc8641d\_simple.soc.cpu[0]**). This has the same effect as if you had executed the following two commands:

```
simics> new-branch-recorder mpc8641d_simple.soc.cpu_0__branch_recorder 2  
physical  
simics> mpc8641d_simple.soc.cpu[0].attach-branch-recorder 2  
mpc8641d_simple.soc.cpu_0__branch_recorder
```

These commands create a branch recorder and attach it to the **mpc8641d\_simple.soc.cpu[0]** processor. All branches taken by **mpc8641d\_simple.soc.cpu[0]** will now be recorded in **mpc8641d\_simple.soc.cpu\_0\_\_branch\_recorder**.

You need to use these separate commands instead of **start-instruction-profiling** if you want a branch recorder with a particular name, or if you want to record virtual instead of physical addresses.

A branch recorder remembers the source and destination address of each branch, as well as the number of times the branch has been taken. It does *not* remember in which order the branches happened, so it cannot be used to reconstruct an execution trace (if you want that, you have to use the **trace** module, which is slower and generates much more profiling data). There is however enough information to compute a number of interesting statistics. To get a list of what the branch recorder can do, type:

```
simics> mpc8641d_simple.soc.cpu_0__branch_recorder.address-profile-info
mpc8641d_simple.soc.cpu_0__branch_recorder has 6 address profiler views:
View 0: execution count
        64-bit physical addresses, granularity 4 bytes
View 1: branches from
        64-bit physical addresses, granularity 4 bytes
View 2: branches to
        64-bit physical addresses, granularity 4 bytes
View 3: interrupted execution count
        64-bit physical addresses, granularity 4 bytes
View 4: exceptions from
        64-bit physical addresses, granularity 4 bytes
View 5: exceptions to
        64-bit physical addresses, granularity 4 bytes
```

An address profiler is an object whose data can be meaningfully displayed as a count for each address interval. The output of the last command indicates that **mpc8641d\_simple.soc.cpu\_0\_\_branch\_recorder** is an address profiler with six separate *views*; i.e., there are six separate ways of displaying its data as counts for each four-byte interval.

View 0 is the execution count per instruction, conveniently represented as one count per four-byte address interval since the instructions of the simulated machine (a PPC in this case) are all four bytes long and aligned on four-byte boundaries. Views 1 and 2 count the number of times the processor has branched from and to each instruction, except when the branch is caused by an exception (or exception return); those branches are counted separately by views 4 and 5. View 3, finally, counts the number of times an instruction was started but not completed because an exception occurred.

When you are done recording for the moment, type:

```
simics> mpc8641d_simple.soc.cpu[0].detach-branch-recorder 2
mpc8641d_simple.soc.cpu_0__branch_recorder
```

to detach the branch recorder; it will stop recording new branches, but the already collected branches will remain in the branch recorder (until you type **mpc8641d\_simple.soc.cpu\_0\_\_branch\_recorder.clean** at the prompt). You can reattach it again at any time:

```
simics> mpc8641d_simple.soc.cpu[0].attach-branch-recorder mpc8641d_simple.soc
```

Section 6.3 explains how to access the recorded data.

### 6.1.1 Virtual Instruction Profiling

Branch recorders can record virtual instead of physical addresses; just say so when you create them:

```
simics> new-branch-recorder ls_profile virtual
```

Once it has been created, the new branch recorder object behaves just the same as a physical profiler would, except for one thing: when you want a physical profile, you typically expect the profiler to collect statistics from the whole system, but when you want a virtual profile, you probably are interested in one process (or the kernel) only.

To be consistent with the name of the branch recorder we just created, let us collect a virtual profile of a run of the `ls` program. The problem is to have the branch recorder attached when `ls` is running, and detached when other processes are running. This can be achieved using the OS awareness framework by registering notification callbacks in the software tracker. The example in section 9.9 shows how to use the software tracker to ensure that a branch recorder is active for a certain process.

## 6.2 Data Profiling

---

**Note:** Data profiling requires that Simics is started in stall mode.

---

In addition to recording branches, Simics can record memory reads and writes by physical address—all of it simultaneously if you like. Start it like this:

```
simics> mpc8641d_simple.soc.cpu[0].add-memory-profiler read
[mpc8641d_simple.soc.cpu[0]] New profiler added for memory read: 2
mpc8641d_simple.soc.cpu[0]_read_mem_prof
```

This creates a new data profiler called `mpc8641d_simple.soc.cpu_0__read_mem_prof`, which will keep track of all reads performed by `mpc8641d_simple.soc.cpu[0]` until you detach it:

```
simics> mpc8641d_simple.soc.cpu[0].remove-memory-profiler read
```

Like branch recorders, data profilers retain their recorded data after being detached, and may be reattached later. They may also be attached to another processor, even while they are still attached to the first one.

And, like branch recorders, data profilers are also address profilers. This means that while they may be different under the hood, they present their data the same way:

```
simics> mpc8641d_simple.soc.cpu_0__read_mem_prof.address-profile-info
mpc8641d_simple.soc.cpu_0__read_mem_prof has 1 address profiler view:
View 0: data profiler
        64-bit physical addresses, granularity 32 bytes
```

However, unlike branch recorders, data profilers can only record physical addresses. Section 6.3 explains how to access the recorded data.

Cache models provide profiling on cache hits and misses (see chapter 7 for more information).

## 6.3 Examining the Profile

Examining the collected profile is the same for any address profiler, no matter if it is really a data profiler, a branch recorder, or something else. For the examples, we use the branch recorder from section 6.1.

Run a few million instructions or so with the branch profiler attached to the processor to get some data to look at, and then type:

```
simics> mpc8641d_simple.soc.cpu_0__branch_recorder.address-profile-data
View 0 of mpc8641d_simple.soc.cpu_0__branch_recorder: execution count
64-bit physical addresses, profiler granularity 4 bytes
Each cell covers 24 address bits (16 Megabytes).
```

column offsets:	0x1000000*	0x0	0x1	0x2	0x3	0x4	0x5	0x6	0x7
0x0000000000000000:	1775M	.	.	.	.	.	.	.	.
0x0000000008000000:	.	.	.	.	.	.	.	.	.
0x0000000010000000:	.	.	.	.	.	.	.	.	.
0x0000000018000000:	.	.	.	.	.	.	.	.	.
0x0000000020000000:	.	.	.	.	.	.	.	.	.
0x0000000028000000:	.	.	.	.	.	3738	.	168918	.
0x0000000030000000:	.	.	.	.	.	.	.	.	.
0x0000000038000000:	.	.	.	.	.	.	.	.	.
0x0000000040000000:	.	.	.	.	.	.	.	.	.
0x0000000048000000:	.	.	.	.	.	.	.	.	.
0x0000000050000000:	.	.	.	.	.	.	.	.	.
0x0000000058000000:	.	.	.	.	.	.	.	.	.
0x0000000060000000:	.	.	.	.	.	.	.	.	.
0x0000000068000000:	.	.	.	.	.	.	.	.	.
0x0000000070000000:	.	.	.	.	.	.	.	.	.
0x0000000078000000:	.	.	.	.	.	.	.	158M	.



```
1934192097 (1934M) counts shown. 0 not shown.
```

(View 0 is the default, so we did not have to specify it explicitly.) This gives us an overview of the address space. Since we did not specify what address interval we wanted to see, we got the smallest interval that contained all counts. By giving arguments to **address-profile-data**, we can zoom in on the interesting part where almost all the action is. A few orders of magnitude closer, it looks like this:

```
simics> mpc8641d_simple.soc.cpu_0__branch_recorder.address-profile-data 2
start = 0x0 stop = 0x1000000
View 0 of mpc8641d_simple.soc.cpu_0__branch_recorder: execution count
64-bit physical addresses, profiler granularity 4 bytes
Each cell covers 17 address bits (128 kilobytes).
```

```
column offsets:
          0x10000*   0x0    0x2    0x4    0x6    0x8    0xa    0xc    0xe
-----
0x0000000000000000:   176M 12405k 22285k 881996 613367 101074    875 17514
0x0000000000010000:   4379k 423593 543383    3049 20195 16316 2270k    746
0x0000000000020000:   22154 6355k      .      .      . 1511M 36732k      .
0x0000000000030000:      .      .      .      .      .      .      . 5161
0x0000000000040000:      .      .      .      .      .      .      .      .
0x0000000000050000:      .      .      .      .      .      .      .      .
0x0000000000060000:      .      .      .      .      .      . 78984      .
0x0000000000070000:  1230k      .      .      . 53      .      .      .
0x0000000000080000:      .      .      .      . 1313      .      .      .
0x0000000000090000:      .      .      .      .      .      .      .      .
0x00000000000a0000:      .      .      .      .      .      .      .      .
0x00000000000b0000:      .      .      .      .      .      .      .      .
0x00000000000c0000:      .      .      .      .      .      .      .      .
0x00000000000d0000:      .      .      .      .      .      .      .      .
0x00000000000e0000:      .      .      .      .      .      .      .      .
0x00000000000f0000:      .      .      .      .      .      .      .      .
0x0000000000100000:      .      .      .      .      .      .      .      .
```

```
1775888101 (1775M) counts shown. 158303996 (158M) not shown.
```

Here, we have zoomed in on the address space from address 0x0 to address 0x1000000. It is also possible to specify an address you want to center on and the size of the address space range you are interested in; see the *Simics Reference Manual* or the online help for **address-profile-data**.

If you want to see the current numbers every time you disassemble code, use the command **aprof-views** to select the views you want:

```
simics> mpc8641d_simple.soc.cpu[0].aprof-views 2
add = mpc8641d_simple.soc.cpu_0__branch_recorder view = 0
simics> si 5
[mpc8641d_simple.soc.cpu[0]] v:0x900038d8 p:0x0000038d8 0 mtmsr r7
[mpc8641d_simple.soc.cpu[0]] v:0x900038d8 p:0x0000038d8 0 mtmsr r7
[mpc8641d_simple.soc.cpu[0]] v:0x900038d8 p:0x0000038d8 0 mtmsr r7
[mpc8641d_simple.soc.cpu[0]] v:0x900038d8 p:0x0000038d8 0 mtmsr r7
[mpc8641d_simple.soc.cpu[0]] v:0x900038d8 p:0x0000038d8 0 mtmsr r7
```

Note that since the disassembly printed by commands such as **c** and **si** are the instruction just about to execute, the statistics reflect the way things were immediately before that instruction executed.

For many real-world profiling tasks, the above methods of looking at profile data are inadequate and lacking in high-level structure. But it can be used as a foundation for building more advanced custom-built tools. One tool that is built using these primitives is the code coverage command described in chapter 5. See the Profiling API section in the *Simics Reference Manual* for more details.

## Chapter 7

# Cache Simulation

### 7.1 Introduction to Cache Simulation with Simics

By default, Simics does not model any cache system. It uses its own memory system to achieve high speed simulation, and modeling a hardware cache model would only slow it down. Simics exposes, however, the flow of memory transactions coming from the processor, and thus allows users to write tracing tools and collect statistics on the memory behavior of their simulations.

Additionally, Simics lets user-written *timing models* control how long each memory transaction takes. *Stalling* transactions, as it is called in Simics, helps improving the timing accuracy of the simulation, as compared to a real system.

These properties make Simics very suitable for various types of cache simulation:

#### Cache Profiling

The goal is to gather information about the cache behavior of a system or an application. Unless the application runs on multiprocessors, takes a lot of interrupts or runs a lot of system-level code, the timing of the memory operations is often irrelevant, thus no stalling is necessary. The memory interfaces are used to be informed of all transactions sent by the processor.

Note that this type of simulation *does not change* the execution of the target program. It could be done by using Simics as a simple memory transaction trace generator, and then computing the cache state evolution afterwards. However, doing the cache simulation at the same time as the execution enables a number of optimizations that Simics models make good use of.

#### Cache Timing

The goal is to study the timing behavior of the transactions, in which case a transaction to memory should take much more time than, for example, a transaction to an L1 cache. This is useful when studying interactions between several CPUs, or to estimate the number of cycles per instruction (CPI) of an application. Simics models can be used for such a simulation.

This type of simulation *modifies* the execution, since interrupts and multi-processor interaction will be influenced by the timing provided by the cache model. However,

unless the target program is not written properly, the execution will always be correct, although different from the execution obtained without any cache model.

### Cache Content Simulation

It is possible to change Simics coherency model by allowing a cache model to contain data that is different from the contents of the memory. Such a model needs to properly handle the memory transactions as it must be able to change the values of loads and stores.

Note that this kind of simulation is difficult to do and requires a well-written, bug-free cache model, since it can prevent the target program from executing properly.

Simics comes with a cache model called **g-cache**, which allows for the first two types of cache simulation: it handles one transaction at a time in a flat way: all needed operations (copy-back, fetch, etc.) are performed in order and at once. The cache returns the sum of the stall times reported for each operation.

Before going further and describing g-cache in more details, a few things should be mentioned:

- When simulating with caches, it is imperative to start Simics with the *-stall* flag to get correct cache statistics. It is possible to start Simics without it, but no transactions will then be stalled, and all transaction may not be visible to the cache.
- Although it is often said that Simics models a processor, such as a PowerPC 440GP or an Intel<sup>®</sup> Core<sup>™</sup>2 processor, Simics is first and foremost an *instruction-set* simulator, not a processor simulator. Transactions coming out of a real Intel<sup>®</sup> Core<sup>™</sup>2 processor have already gone through the L1 and L2 caches, so they consist mainly of cache misses to be fetched from memory. On contrary, in Simics, all transactions issued from the processor core are visible to the cache model.
- For simplicity and performance, Simics does not model incoherence. In Simics, the memory is *always* up to date with the latest CPU and device transactions. This property holds even when doing cache simulation with Simics standard model, as it does not contain any data, only cache line status information.

## 7.2 Simulating a Simple Cache

Caches models are not currently well integrated with the component system that Simics uses for other devices. For that reason, users are required to create caches and connect them by hand. This approach offers, on the other hand, total control on the configuration process.

Adding a cache to a configuration is easily done with a small Python script, here, adapted for a standard Firststeps configuration:

```
from simics import *

cpu = conf.mpc8641d_simple.soc.cpu[0]
cache = pre_conf_object('cache', 'g-cache',
```

```

config_line_number = 256,
config_line_size = 32,
config_assoc = 1,
config_virtual_index = 0,
config_virtual_tag = 0,
config_replacement_policy = 'random',
penalty_read = 0,
penalty_write = 0,
penalty_read_next = 0,
penalty_write_next = 0,
cpus = cpu)

SIM_add_configuration([cache], None)

cpu.physical_memory.timing_model = conf.cache

```

The script starts by creating a preliminary cache object with a number of parameters: 256 lines of 32 bytes each, with no associativity, physical index and tags and random replacement policy. No stalling is enabled, as all penalties are 0. All the cache parameters are described in more details further in this chapter.

The preliminary cache object is added to the current configuration, and then connected as a listener object on the memory-space representing the physical memory of the processor in the system. From now on, accesses to main memory will go through the cache and cache hits/misses will be simulated.

If you run the simulation for a while, you will get information about the cache with the **cache.status** and **cache.statistics** commands.

---

**Note:** For performance reasons, instruction fetches are not sent to the caches unless you explicitly ask for it. Refer to section 7.4 for more information.

---

## 7.3 A More Complex Cache System

Using **g-cache**, we can simulate more complex cache systems. Let us consider the structure described by figure 7.1.

What we want to simulate is a system with separate instruction and data caches at level 1 backed by a level 2 cache. We will connect this memory hierarchy to the Firststeps machine as previously. The dotted components in the diagram represent elements that are introduced in Simics to complete the simulation. Let us have a look at these components:

### **id-splitter**

This class is used by Simics to separate instruction and data accesses and send them to separate L1 caches.

### **splitter**

We might receive, although this is unlikely, accesses that crosses a cache-line boundary. To avoid that, we connect two splitters before the caches. The splitters will let

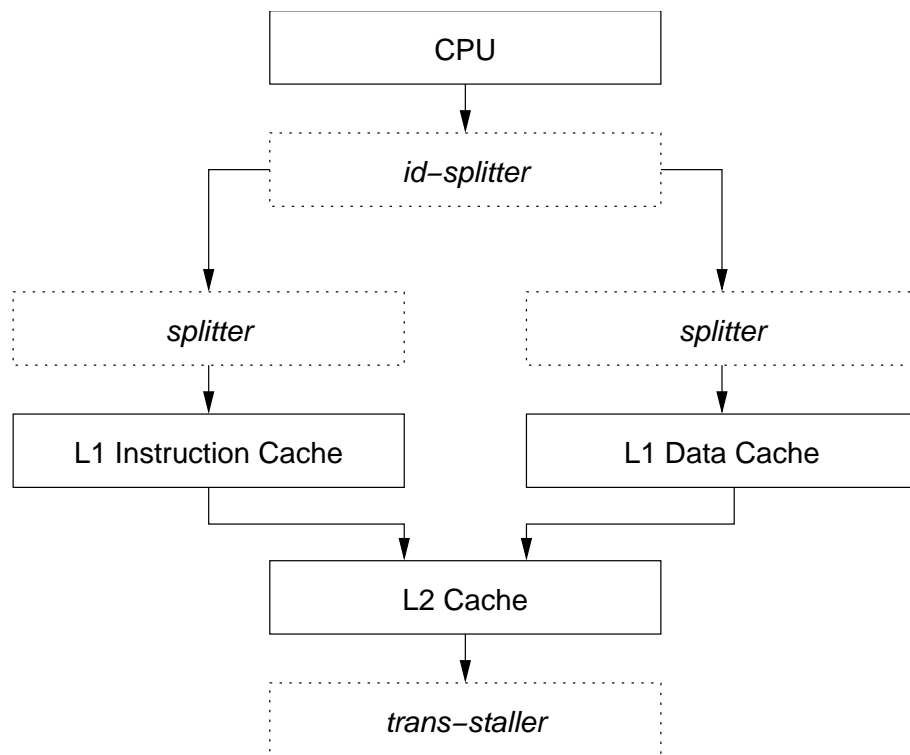


Figure 7.1: A More Complex Cache System

uncacheable and correctly aligned accesses go through untouched, whereas others will be split in two accesses.

#### **trans-staller**

The **trans-staller** is a class that simulates memory latency. It will stall all accesses by a fixed amount of cycles.

In terms of script, this configuration gives us the following:

```
from simics import *

cpu = conf.mpc8641d_simple.soc.cpu[0]

# transaction staller to represent memory latency
staller = pre_conf_object('staller', 'trans-staller',
                          stall_time = 200)

# L2 cache: 512Kb write-back
l2c = pre_conf_object('l2c', 'g-cache',
                      config_line_number = 4096,
                      config_line_size = 128,
                      config_assoc = 8,
                      config_virtual_index = 0,
                      config_virtual_tag = 0,
                      config_write_back = 1,
                      config_write_allocate = 1,
                      config_replacement_policy = 'lru',
                      penalty_read = 10,
                      penalty_write = 10,
                      penalty_read_next = 0,
                      penalty_write_next = 0,
                      cpus = cpu,
                      timing_model = staller)

# instruction cache: 16Kb
ic = pre_conf_object('ic', 'g-cache',
                     config_line_number = 256,
                     config_line_size = 64,
                     config_assoc = 2,
                     config_virtual_index = 0,
                     config_virtual_tag = 0,
                     config_replacement_policy = 'lru',
                     penalty_read = 3,
                     penalty_write = 0,
                     penalty_read_next = 0,
                     penalty_write_next = 0,
```

```

        cpus = cpu,
        timing_model = l2c)

# data cache: 16Kb write-through
dc = pre_conf_object('dc', 'g-cache',
                    config_line_number = 256,
                    config_line_size = 64,
                    config_assoc = 4,
                    config_virtual_index = 0,
                    config_virtual_tag = 0,
                    config_replacement_policy = 'lru',
                    penalty_read = 3,
                    penalty_write = 3,
                    penalty_read_next = 0,
                    penalty_write_next = 0,
                    cpus = cpu,
                    timing_model = l2c)

# transaction splitter for instruction cache
ts_i = pre_conf_object('ts_i', 'trans-splitter',
                      cache = ic,
                      timing_model = ic,
                      next_cache_line_size = 64)

# transaction splitter for data cache
ts_d = pre_conf_object('ts_d', 'trans-splitter',
                      cache = dc,
                      timing_model = dc,
                      next_cache_line_size = 64)

# instruction-data splitter
id = pre_conf_object('id', 'id-splitter',
                    ibranch = ts_i,
                    dbranch = ts_d)

SIM_add_configuration([staller, l2c, ic, dc, ts_i, ts_d, id], None)

cpu.physical_memory.timing_model = conf.id

```

The hierarchy is linked by chaining the timing models one after another. Once all objects are created, the only thing left is to connect the top of the hierarchy, the **id-splitter** `id`, to the main memory.

Stalling has been enabled in this configuration, as some of the cache penalties have been set. In that particular case, the *\_next* penalties are not used, and the next level cache reports penalties when they are accessed. For example, a read hit in L1 would take 3 cycles; a read



miss that goes to memory would take  $3 + 10 + 200 = 213$  cycles if no copy-back is performed in the L2 cache.

## 7.4 Caches Models and Instruction Fetches

For performance reasons, instruction fetches are not sent to the cache models by default.

instruction fetches can be activated for each processor with the `<cpu>.instruction-fetch-mode` command. It can take several values:

### *no-instruction-fetch*

No instruction fetches are sent to the memory hierarchy interface.

### *instruction-cache-access-trace*

An instruction fetch is sent every time a different cache line is accessed by the processor. The size of the cache line is set by the processor attribute *instruction-fetch-line-size*.

This option is meant to be used for cache simulation where successive accesses to the same cache line do not modify the cache state.

### *instruction-fetch-trace*

All instruction fetches are sent to the memory hierarchy interface. This option is often implemented as *instruction-cache-access-trace* with a line size equal to the size of one instruction.

This option is meant to provide a complete trace of fetch transactions.

Note that the `<cpu>.instruction-fetch-mode` is only available when Simics is running in *stall* mode. Stall mode is enabled with the `-stall` command line flag to Simics or by changing to stall mode in the preferences and then restart.

Finally, instruction fetch transactions are not generated by all processor models. The section 8 contains a summary of which features are available on which models.

## 7.5 Workload Positioning and Cache Models

In the previous examples, it is worth noticing that connecting the caches to the memory-space can be done separately from defining the caches. It is thus possible to connect and disconnect the caches at any time during the simulation: for example, the operating system boot and workload setup can be done with Simics in normal mode to create a checkpoint. The checkpoint is then reloaded in `-stall` mode with cache simulation enabled.

To get decent cache statistics, it is important to run at least a few million instructions to warm up the caches before actually starting to do measurements. Note that this is only a rough advice; the precise warm-up time needed will depend on the cache model and the workload.

## 7.6 Using g-cache

Let us have a more detailed look at **g-cache**. It has the following features:

- Configurable number of lines, line size and associativity. Note that the line size must be a power of 2, and that the number of lines divided by the associativity must be a power of two.
- Physical/virtual index and tag.
- Configurable write allocate/write back policy.
- Random, true LRU or cyclic replacement policies. It is easy to add new replacement policies.
- Sample MESI protocol.
- Support for several processors connected to one cache.
- Configurable penalties for read/write accesses to the cache, and read/write accesses initiated by the cache to the next level cache.
- Cache miss profiling.

Transactions are handled one at a time; all operations are done in order, at once, and a total stall time is returned. The transaction is not reissued afterward. Here's a short description of the way **g-cache** handles a transaction (we assume a write-back, write allocate cache):

- If the transaction is uncacheable, **g-cache** ignores it.
- If the transaction is a read hit, **g-cache** returns *penalty\_read* cycles of penalty.
- If the transaction is a read miss, **g-cache** asks the replacement policy to provide a cache line to allocate.
- The new cache line is emptied. If necessary, a copy-back transaction is initiated to the next level cache. In this case, a penalty of *penalty\_write\_next* is counted, added to the penalty returned by the next level.
- The new data is fetched from the next level, incurring *penalty\_read\_next* cycles penalty added to the penalty returned by the next level.
- The total penalty returned is the sum of *penalty\_read*, plus the penalties associated with the copy-back (if any), plus the penalties associated with the line fetch.

Note the usage of *penalty\_read/write* and *penalty\_read/write\_next*: a write-through cache would always take a *penalty\_write\_next* penalty independently of the fact that a write is a hit or a miss, but **g-cache** always reports hits and misses according to the line look-up, not based on the fact that a transaction is propagated to the next level.

## 7.7 Understanding g-cache Statistics

The following statistics are available in a **g-cache**:

### **Total number of transactions**

Count of all transactions received by the cache, including all transactions listed below.

### **Device data read/write**

Number of transactions, e.g., DMA transfers, performed by devices against the memory-space to which the cache is connected. Device accesses are otherwise ignored by the cache and passed as-is to the next level cache.

### **Uncacheable data read/write, instruction fetch**

Number of uncacheable transactions performed by the processor. Uncacheable transactions are otherwise ignored by the cache and passed as-is to the next level cache.

### **Data read transactions**

Cacheable read transactions counted by the cache.

### **Data read misses**

Cacheable read transactions that missed in the cache.

### **Data read hit ratio**

$1 - (\text{cacheable read misses} / \text{cacheable read transactions})$

### **Instruction fetch transactions**

Cacheable instruction fetch transactions counted by the cache.

### **Instruction fetch misses**

Cacheable instruction fetch transactions that missed in the cache.

### **Instruction fetch hit ratio**

$1 - (\text{cacheable instruction fetch misses} / \text{cacheable instruction fetch transactions})$

### **Data write transactions**

Cacheable write transactions counted by the cache.

### **Data write misses**

Cacheable write transactions that missed in the cache. This is not directly related to the number of transactions sent to the next level cache, which also depends on the write allocation and write-back policies selected for the cache.

### **Data write hit ratio**

$1 - (\text{cacheable write misses} / \text{cacheable write transactions})$

### **Copy-back transactions**

Copy-back transactions performed by the cache to flush modified cache lines.

**Lost Stall Cycles**

Number of cycles the processor model should have stalled, but did not, because the memory transaction was not allowed to. This number represents how the model limitations—in this case, non-stallable transactions—are preventing timing from being fully taken into account.

**7.8 Speeding up g-cache simulation**

By default, **g-cache** will try to use the simulator's translation caches (STCs) to minimize the number of transactions that it has to handle while still providing correct statistics and behavior. To describe them briefly, the STCs cache addresses and access counters with no-side effects to avoid going through the memory system for every operation.

**Simulating a data cache**

If you are only interested in data accesses, **g-cache** can use the Data STC and still provide correct statistics. To allow **g-cache** to use the Data STC, you should set the *penalty\_read* and *penalty\_write* to 0 (so that cache hits do not take any penalty). **g-cache** will then allow cache hit transactions to be saved in the Data STC and use its internal counters to report a correct number of total transactions, and thus correct ratios.

Note that when using the Data STC counters, **g-cache** can not determine to which memory space the accesses reported by the DSTC belong to, so you need to connect **g-cache** to all the memory spaces to which the processor is talking. In practice, the processor often talks to one main memory space and nothing special needs to be done. Sun's UltraSPARC machines, however, have separate physical memory and physical I/O spaces. The cache should then be connected to both of them. Another way of solving this problem is to connect a small module that will prevent accesses to other memory spaces from being cached in the Data STC.

**Simulating an instruction cache**

When simulating an instruction cache, **g-cache** is able to use the Instruction STC to speed up the simulation and report the number of instruction misses, but it will not report the correct number of total transactions. If you wish to have a correct total amount of instruction fetches, you need to disable ISTC usage at the command line with the **istc-disable** command.

You can always prevent **g-cache** from using the STCs if you encounter one of the limitations mentioned later in this chapter, by setting the *config\_block\_STC* attribute of the cache to 1.

**7.9 Cache Miss Profiling**

**g-cache** can use Simics's data profiling support to profile cache misses. You can use the **add-profiler** and **remove-profiler** commands to add and remove profiler to the cache for a specific type of events.

**g-cache** can profile the following type of events:

- number of data read misses per virtual address, physical address, or instruction
- number of data write misses per virtual address, physical address, or instruction
- number of instruction fetch misses per virtual or physical address

For example, if you would like to see which parts of the code are responsible for read and write misses, you could create profilers that count read and write misses per instruction. (This example assumes that your cache is called **dc**.)

```
simics> dc.add-profiler type = data-read-miss-per-instruction
[dc] New profiler added for data-read-miss-per-instruction: 2
dc_prof_data-read-miss-per-instruction
simics> dc.add-profiler type = data-write-miss-per-instruction
[dc] New profiler added for data-write-miss-per-instruction: 2
dc_prof_data-write-miss-per-instruction
```

This creates two profiler objects and attaches them to the cache object. The profilers are initially empty, so we have to run for a while to give them time to collect some interesting data:

```
simics> c 10_000_000
[cpu0] v:0x00000000000002590 p:0x0000000003c7e590 sll %i0, 1, %o1
```

Now, we can ask either profiler what data it has gathered:

```
simics> dc_prof_data_read_miss_per_instruction.address-profile-data
View 0 of dc_prof_data_read_miss_per_instruction: dc prof: data-read-miss-per-instruction
64-bit virtual addresses, profiler granularity 4 bytes
Each cell covers 2 address bits (4 bytes).
```

```
column offsets:
          0x1*   0x00   0x04   0x08   0x0c   0x10   0x14   0x18   0x1c
-----
0x00000000000002480:      .      .      . 23331      .      .      .      .
0x000000000000024a0:      .      . 15492    545      .      .    90      .
0x000000000000024c0:      .      .      .   112      .      .      .      .
0x000000000000024e0:      .      .      .      .      .      .      .      .
0x00000000000002500:      .      .      .      .      .      .      .      .
0x00000000000002520:      .      .      .      .      .      .      .      .
0x00000000000002540:      .      .      .      .      .      .      .      .
0x00000000000002560:      .      .      .      .      .      .      .      .
0x00000000000002580:      .      .      .      .      .      .      .      .
0x000000000000025a0:      .      .      .      .      .      .      .      .
0x000000000000025c0:      .      .      .      .      .      .      .      .
0x000000000000025e0:      .      .      .      .      .      .    136      .
```

39706 counts shown. 0 not shown.

Since these two profilers are instruction indexed, it also makes sense to display their counts in a disassembly listing:

```
simics> cpu0.aprof-views add = dc_prof_data_read_miss_per_instruction
simics> cpu0.aprof-views add = dc_prof_data_write_miss_per_instruction
simics> cpu0.disassemble %pc 32
v:0x0000000000002590 p:0x0000000003c7e590    0 0  sll %i0, 1, %o1
v:0x0000000000002594 p:0x0000000003c7e594    0 0  ldub [%o1 + %o2], %o1
v:0x0000000000002598 p:0x0000000003c7e598    0 0  and %i0, %o1, %o1
v:0x000000000000259c p:0x0000000003c7e59c    0 0  sll %o1, 16, %i0
v:0x00000000000025a0 p:0x0000000003c7e5a0    0 0  sra %i0, 16, %i0
v:0x00000000000025a4 p:0x0000000003c7e5a4    0 0  jmp1 [%i7 + 8], %g0
v:0x00000000000025a8 p:0x0000000003c7e5a8    0 0  restore %g0, %g0, %g0
v:0x00000000000025ac p:0x0000000003c7e5ac    0 0  jmp1 [%i7 + 8], %g0
v:0x00000000000025b0 p:0x0000000003c7e5b0    0 0  restore %g0, -1, %o0
v:0x00000000000025b4 p:0x0000000003c7e5b4    0 0  illtrap 0
v:0x00000000000025b8 p:0x0000000003c7e5b8    0 0  illtrap 0
v:0x00000000000025bc p:0x0000000003c7e5bc    0 0  illtrap 0
v:0x00000000000025c0 p:0x0000000003c7e5c0    0 0  illtrap 0
v:0x00000000000025c4 p:0x0000000003c7e5c4    0 0  illtrap 0
v:0x00000000000025c8 p:0x0000000003c7e5c8    0 0  save %sp, -96, %sp
v:0x00000000000025cc p:0x0000000003c7e5cc    0 0  sethi %hi(0x41c00), %o0
v:0x00000000000025d0 p:0x0000000003c7e5d0    0 0  add %o0, 840, %o1
v:0x00000000000025d4 p:0x0000000003c7e5d4    0 0  ldub [%o1 + 0], %o0
v:0x00000000000025d8 p:0x0000000003c7e5d8    0 0  subcc %o0, 1, %o0
v:0x00000000000025dc p:0x0000000003c7e5dc    0 0  stw %o0, [%o1 + 0]
v:0x00000000000025e0 p:0x0000000003c7e5e0    0 0  bpos 0x25f0
v:0x00000000000025e4 p:0x0000000003c7e5e4    0 0  mov -1, %i0
v:0x00000000000025e8 p:0x0000000003c7e5e8    0 0  jmp1 [%i7 + 8], %g0
v:0x00000000000025ec p:0x0000000003c7e5ec    0 0  restore %g0, %g0, %g0
v:0x00000000000025f0 p:0x0000000003c7e5f0    0 0  sethi %hi(0x4f000), %o0
v:0x00000000000025f4 p:0x0000000003c7e5f4    0 0  add %o0, 616, %o0
v:0x00000000000025f8 p:0x0000000003c7e5f8   136 0  ldub [%o0 + 0], %o2
v:0x00000000000025fc p:0x0000000003c7e5fc    0 0  add %o2, 1, %o1
v:0x0000000000002600 p:0x0000000003c7e600    0 0  stw %o1, [%o0 + 0]
v:0x0000000000002604 p:0x0000000003c7e604    0 0  ldub [%o2 + 0], %o2
v:0x0000000000002608 p:0x0000000003c7e608    0 0  and %o2, 255, %i0
v:0x000000000000260c p:0x0000000003c7e60c    0 0  jmp1 [%i7 + 8], %g0
```

In the listing above, we see that in the 32 instructions following the current program counter, one load instruction is responsible for 136 read misses, and no instructions have caused any write misses.

For more on getting information out of profilers, refer to chapter 6.

## 7.10 Using g-cache with Several Processors

Support for several processors talking to one cache is integrated in **g-cache**. You just need to specify the list of CPUs connected to the cache in the *cpus* attribute. Note that it is possible for the cache to use the STCs as described above even with several processors.

You can use the sample MESI protocol to connect several caches in a multiprocessor system. You need to provide the cache with a list of the other caches snooping on the bus using the *snoopers* attribute. If you have higher-level caches that are not snooping on the bus, you need to set the *higher\_level\_caches* attribute so that invalidation is done properly. When setting up a cache system using cache components, both the *snoopers* and the *higher\_level\_caches* attributes will be set automatically *if* a memory timer component is used to simulate the bus and memory. Note that the sample MESI protocol was written to handle simple cases such as several L1 write-through caches with L2 caches synchronizing via MESI. To model more complex protocol, you will need to modify **g-cache**.

If you use LRU replacement with several processors, you may have problems with the way Simics schedules processor execution. You may want to lower the CPU switch time, using the command **cpu-switch-time**, from its default value to prevent accesses “in the future” from changing the way LRU replacement is behaving. The *Understanding Simics Timing* application note describes how multiprocessor simulation and time quantum operates.

## 7.11 Simulating Private Caches

Since a cache is connected to a memory space through its *timing\_model* attribute, the cache is presented with all memory operations that pass through the memory space. Usually, in Simics, cores in the same processor or SMP multiprocessor systems are initially configured to use the same main memory-space, so connecting a cache to this memory-space will allow the cache to see transactions from all processors at once.

When simulating a multi processor machine with private caches, this is not the desired scenario. The solution is to insert extra memory spaces for each processor, and let each processor send memory transactions only to a private memory space. These extra memory spaces are then ideal connection points for private caches, since it is known that all memory transactions will come from a particular processor. In some target configurations memory spaces that can fill this function already exist. To determine if this is the case, it is possible to look at the *physical\_memory* attribute of the processors. This attribute determines to which object the processor should send its memory transactions. If this attribute points to the same object for all processors, extra memory spaces are needed to accommodate private cache simulation.

Below is an example that sets up two private level 1 caches and a shared level 2 cache. Note especially how the two extra memory spaces are created and configured and the use of the *cpus* and *snoopers* attributes in the caches. This example is based on the MPC8641 Firststeps machine, configured with two cores.

```
from simics import *

# the two cores
```

```

cpu0 = conf.mpc8641d_simple.soc.cpu[0]
cpu1 = conf.mpc8641d_simple.soc.cpu[1]

# the memory-space the two cores use as main memory
phys_mem = cpu0.physical_memory

# shared cache between the two cores
cacheS = pre_conf_object('cache_shared', 'g-cache',
                        cpus = [cpu0, cpu1],
                        config_line_number = 256,
                        config_line_size = 32,
                        config_assoc = 1,
                        config_virtual_index = 0,
                        config_virtual_tag = 0,
                        config_replacement_policy = 'random',
                        penalty_read = 0,
                        penalty_write = 0,
                        penalty_read_next = 0,
                        penalty_write_next = 0)

# private cache for cpu0
cache0 = pre_conf_object('cache_cpu0', 'g-cache',
                        cpus = cpu0,
                        config_line_number = 256,
                        config_line_size = 32,
                        config_assoc = 1,
                        config_virtual_index = 0,
                        config_virtual_tag = 0,
                        config_replacement_policy = 'random',
                        penalty_read = 0,
                        penalty_write = 0,
                        penalty_read_next = 0,
                        penalty_write_next = 0,
                        timing_model = cacheS)

# private cache for cpu1
cache1 = pre_conf_object('cache_cpu1', 'g-cache',
                        cpus = cpu1,
                        config_line_number = 256,
                        config_line_size = 32,
                        config_assoc = 1,
                        config_virtual_index = 0,
                        config_virtual_tag = 0,
                        config_replacement_policy = 'random',
                        penalty_read = 0,

```



```

        penalty_write = 0,
        penalty_read_next = 0,
        penalty_write_next = 0,
        timing_model = cacheS)

# set up MESI protocol between cache0 and cache1
cache0.snoopers = [cache1]
cache1.snoopers = [cache0]

# two separate memory spaces pointing only at phys_mem
mem0 = pre_conf_object('cpu0_mem', 'memory-space',
                       default_target = [phys_mem, 0, 0, phys_mem],
                       timing_model = cache0)
mem1 = pre_conf_object('cpu1_mem', 'memory-space',
                       default_target = [phys_mem, 0, 0, phys_mem],
                       timing_model = cache1)

SIM_add_configuration([cache0, cache1, cacheS, mem0, mem1], None)

# point each core at its private main memory space, with cache already
# configured. Done!
cpu0.physical_memory = conf.cpu0_mem
cpu1.physical_memory = conf.cpu1_mem

```

## 7.12 g-cache Limitations

Virtually tagged caches are usually informed of the state of the MMU in order to immediately invalidate lines whose virtual-to-physical mapping changes. **g-cache** is not MMU-aware, so when looking for a hit with virtual tags, it tries to match both the virtual tag and the physical tag. This approach prevents cache accesses from triggering a hit on lines with invalid translations, but it makes it possible for some invalid mappings to be present in the cache (and shown by the **status** command for example).

Some special instructions (atomic instructions in particular) can cause the STC counters to be off by about one memory access per million, which influences the total number of transactions reported by the cache. The UltraSPARC architecture should be free of this bug, while others may still trigger it in some circumstances. The workaround is to avoid using the STC if a very precise total transaction count is needed.

**g-cache** does not understand control transactions like cache flushes and cache line locking.

## Chapter 8

# Processor-specific Features and Limitations

All types of cache and profiling features are not supported by all processor types. Features available in standard Simics functional models are listed below. The feature set and availability may be different for third-party processor models or for cycle accurate or cycle approximate models.

### Execution Trace (Trace)

All instructions executed on the simulated processors can be traced by a Simics extension. An example extension, the trace module, is included that traces instructions as well as data accesses, and either prints them to the simulator console or to a binary file for off-line processing.

### Instruction Fetch Mode (IFetch)

In instruction fetch mode, one instruction fetch is sent out to a user defined memory hierarchy or trace module for each instruction executed. Instruction fetch mode can be emulated for processors with a fixed sized instruction length by setting the cache-line size to the length of the instructions.

### Cache Access Instruction Fetch Mode (Cache IFetch)

In cache access mode, instruction fetches for branch targets and cache-line crossings are sent to a user defined memory hierarchy or trace module. The size of cache-lines is configurable on a per processor basis.

### Stall on Instruction Fetches (IStall)

Processors running in *stall* mode allow a user defined memory hierarchy to specify how long time memory accesses due to instruction fetches take.

### Stall on Data Accesses (DStall)

Processors running in *stall* mode allow a user defined memory hierarchy to specify how long time memory accesses due to data read and writes take. For instructions that generate multiple memory references, only the first access can be stalled.

### Instruction Profiling (IProf)

The instruction profiler collects an exact count of instruction executed on each virtual

or physical address. Virtual address profiling requires process tracking for the target OS for separation of the counts for different virtual address spaces.

### Data Profiling (DProf)

The data profiler collects an exact count of data accesses on physical address.

	Trace	IFetch	Cache IFetch	IStall	DStall	IProf	DProf
<b>ARMv5</b>	y	-	-	-	-	-	y
<b>MIPS32</b>	y	y	y	-	y	y	y
<b>MIPS64</b>	y	y	y	-	y	y	y
<b>PPC32</b>	y	y	y	-	y	y	y
<b>PPC64</b>	y	y	y	-	y	y	y
<b>SPARC V8</b>	y	-	-	-	-	-	y
<b>SPARC V9</b>	y	y	y	y	y	y	y
<b>TMS320c64</b>	y	-	-	-	-	-	y
<b>X86</b>	y	y	-	-	y	y	y
<b>X86-64</b>	y	y	-	-	y	y	y

## Chapter 9

# OS Awareness Details

This chapter describes how the OS awareness features of Simics works and how to configure and extend it. The information in this chapter is important if you are creating system configurations and want to configure the software tracker, if you are writing your own software tracker, or if you want to use the OS awareness system directly in Python scripts.

---

**Note:** The OS awareness modules have recently been updated; mainly, the node trees have a slightly different structure. This is unfortunately not yet reflected in this text.

---

### 9.1 Tracker Objects And System Configurations

Every system configuration script that creates a system on which software can run should add a `os_awareness` component to the `software` slot of each top-level component. This is done by running the **create-os-awareness** command and naming the new component by appending `.software` to the system component name. This is how it's done in the standard configuration files (usually called *something-system.include*).

```
load-module os-awareness

create-os-awareness name = $system.software
Created non-instantiated 'os_awareness' component 'test_board.software'
```

Notice how the **create-os-awareness** command creates a non-instantiated component. In order to be able to use the software system the component has to be instantiated first. The standard procedure is to create the `os-awareness` component in the target script before instantiating components. However, if that is not the case, it can be instantiated by running the following **instantiate-components** command

The software component handles user interaction by providing a number of commands like `system.software.track`, but it only contains a single other object. This object is the *tracker* object, an instance of the **software\_tracker** class, which does the heavy lifting of keeping track of the running software.

However, this tracker won't be able to track anything unless it is given a configuration that instructs it how to interpret the system state. This is done by adding *tracker modules* to the tracker object, with appropriate configuration parameters. For instance, if the target system is running Linux 2.6.29, the `linux_tracker` is added to the tracker object with configuration parameters that tells it about various kernel and task offsets. See section 9.8 for more details.

This configuration should be done in the configuration script that configures the software.

```
$system.software.load-parameters tracker-conf.params
```

The *tracker-conf.params* file is a small file that looks something like this:

```
# -*- Python -*-
['linux_tracker', {'ts_offsets': {'active_mm': 180, 'parent': 224,
'sibling_list_head': 236, 'mm': 176, 'pid': 212, 'state': 0,
'next': 152, 'real_parent': 220, 'binfmt': 184,
'children_list_head': 228, 'comm': 456, 'thread_struct': 496,
'prev': 156, 'tgid': 216}, 'kernel_base': 3221225472L,
'ts_next_relative': True}]
```

These files can be created automatically by the OS awareness system. Each tracker comes with some sort of detect command see 2.4 For instance, the **linux-autodetect-settings** command can be run on a booted Linux system and will create a parameters file that can be loaded into the tracker from a configuration script.

```
simics> mpc8641d_simple.software.linux-autodetect-settings linux.params
[mpc8641d_simple.software.tracker info] Autodetecting using 2 of 2 processor(s)
Saved autodetected parameters to linux.params
```

The autodetect command itself is not run from any configuration script. It is meant to be run once manually by the one configuring the system and the generated file is then used from the configuration scripts.

It is possible to use the trackers on a single system running multiple operating systems. In that case each operating system must have dedicated processors and each processor must have it's own physical memory space. Such a system requires some additional configuration steps, that has to be done after the processor objects have been instantiated. First use the **load-module os-awareness** command to load the os-awareness module. Next create one non-instantiated software component for each operating system. This is done with the **create-os-awareness** command. Next set the processors attribute of each software tracker to match that of the operating system it is going to track. Finally instantiate the component using the **instantiate-components** command. The trackers are now ready to be used.

```
simics> load-module os-awareness

simics> create-os-awareness name = system_cmp0.softwareA
```

```

Created non-instantiated 'os_awareness' component 'system_cmp0.softwareA'
simics> create-os-awareness name = system_cmp0.softwareB
Created non-instantiated 'os_awareness' component 'system_cmp0.softwareB'
simics> @conf.system_cmp0.softwareA.processors = [conf.system_cmp0.cpuA]

simics> @conf.system_cmp0.softwareB.processors = [conf.system_cmp0.cpuB]

simics> instantiate-components

```

## 9.2 Tracker Activation

Tracking the software does not come for free. For this reason, the software tracker is disabled by default and has to be enabled to make it track anything or get any information out of it.

Some trackers are able to determine some information about the running software just by enabling it on an already running system, but in most cases there is additional—and sometimes important information—that can only be collected by having the tracker enabled while the software starts. For instance, the Linux software tracker can determine the name and process id of already running processes, but to determine the path to the running binary the tracker needs to be enabled when the process starts. See [section 2.2](#) for information about how to enable the tracker.

Sometimes other parts of Simics, such as visualization tools, will also enable the tracker, and the tracker will be enabled as long as anyone wants it enabled.

## 9.3 The Node Tree

The tracker object—when activated—monitors the machine state, representing it as a tree of nodes. Other objects and scripts can query this tree, and register callbacks that trigger when it changes.

This tree represents the software components that the processors in the system can execute code in, in a hierarchical way. Each node in the hierarchy represents an abstraction that contains all its *child* nodes. For example, one node can represent a process, and its children all the threads in that process. Each node knows what subset of the machine’s processors are currently assigned to it; this makes it possible to see which thread, process, etc. each processor is currently running, and to get callbacks on task switches.

The tree has nodes above the process level as well. For example, a machine running Linux would have a node tree with a root node with two children, one for all kernel processes and one for all user processes. The children of the user node each represent one process; and their children each represent a thread in that process. Nodes also know OS-specific details such as process IDs and program names. [Section 9.8.4](#) has the details for the trackers.

The node tree changes over time, and nodes will be created and destroyed as processes and threads are born and die.

In the interfaces used to interact with the node tree, every node is identified by a *node ID*, which is a simple integer that is unique. The IDs are never reused by the tracker.

Each node has a set of named *properties* that contain information about what the node represents. What properties exist depends on the type of node, but the properties listed below are the standard properties, which all nodes have (except where noted):

**name**

The name of this node. Unlike the ID, it need not be unique.

**extra\_id**

A list of property names that can uniquely identify nodes at the same level. All nodes that have the same parent node must have the same value for *extra\_id*. If the list is empty, the property *name* should be used to identify the node.

**children**

The set of child node IDs (represented as a list). This property is only present in nodes that can have children. Nodes that can have children, but do not currently have any, have an empty list.

**parent**

The ID of the parent of this node. This value never changes during the node's lifetime. This property is not present in the root node.

**processors**

The set of all the processors where the software node is currently running (represented as a list). If it is not currently running on any processor, the list is empty.

**multiprocessor**

True if the node can be running on more than one processor at a time, false if it can be running on at most one processor.

**memory\_space**

For each node which is part of a virtual memory space, or defines such a memory space itself this property contains the ID of the node which defines the memory space.

**context**

The context object for this node, or nil. It can be set by calling the `software` interface's *set\_property* method. See section 9.5 for details.

Note the important concept of a node *running on* a certain processor, as described in the `processors` and `multiprocessor` properties; this simply means that the software represented by the node is currently running on that processor. At any time, the following three laws hold:

1. *For each processor, there exists a node that is running on that processor.* After all, processors are always doing something—sleeping, if nothing else.
2. *If a node is running on a processor, its parent is also running on that processor.* (And, therefore, its grandparent, great grandparent, etc., all the way to the root node.) This is because a child node always represents a subset of its parent. For example, if a processor is currently running a certain thread (a fourth-level node), it is also running the thread's

process (the thread node's parent, a third-level node), and user-space code (the process node's parent, a second-level node), and any code on the machine (the user-space node's parent, the root node).

(This example is from the Linux tracker; see section 9.8.4.)

3. *If two nodes are running on the same processor, then one must be a descendant of the other.* This is a consequence of processors being able to do just one thing at a time. For example, it is not possible for a processor to run more than one thread at a time.

Any node that can be configured to use a tracker has an additional set of properties. This includes the root node, but depending on the tracker configuration, there may be other nodes that also can be configured with a tracker, for instance node representing hypervisor guests.

#### **tracker**

A (name, settings) tuple that defines which tracker module is currently taking care of tracking the software at the given node, or nil if no tracker has been assigned. This property can be set by calling the `software` interface's *set\_property* method. See section 9.8.3.

#### **max\_name\_length**

An integer that describes the maximum length of the *name* property for automatically created nodes under this tracker node. This usually represents the size of the buffer containing the process name used by the tracker. This buffer may be shorter than the full process name. This property is nil or not present if there is no known maximum length.

#### **license\_error**

If the configured tracker can not be used because of license restrictions, this property contains a description of the problem. Otherwise, this property is nil.

Trackers for various hypervisors and operating systems define additional properties to hold process IDs, program file names, and so on. See the documentation for each tracker (section 9.8.4) for details.

## **9.4 The Software Interface**

Almost all interactions with **software-trackers** go via the `software` interface.

### **9.4.1 software\_interface\_t**

```
typedef enum {
    Sim_Prop_Modified,
    Sim_Prop_Added,
    Sim_Prop_Removed
} property_status_change_t;
```



```

SIM_INTERFACE(software) {
    uint64 (*root_node_id)(conf_object_t *NOTNULL obj);
    attr_value_t (*get_node)(conf_object_t *NOTNULL obj, uint64 node_id);
    attr_value_t (*get_subtree)(conf_object_t *NOTNULL obj, uint64 node_id);
    attr_value_t (*get_current_nodes)(
        conf_object_t *NOTNULL obj, conf_object_t *cpu);
    attr_value_t (*notify_create)(
        conf_object_t *NOTNULL obj, uint64 node_id, bool recursive,
        void (*callback)(cbdata_call_t data, conf_object_t *obj,
            conf_object_t *cpu, uint64 node_id),
        cbdata_register_t data);
    attr_value_t (*notify_destroy)(
        conf_object_t *NOTNULL obj, uint64 node_id, bool recursive,
        void (*callback)(cbdata_call_t data, conf_object_t *obj,
            conf_object_t *cpu, uint64 node_id),
        cbdata_register_t data);
    attr_value_t (*notify_property_change)(
        conf_object_t *NOTNULL obj, uint64 node_id, const char *key,
        bool recursive,
        void (*callback)(
            cbdata_call_t data, conf_object_t *obj,
            conf_object_t *cpu, uint64 node_id, const char *key,
            attr_value_t old_val, attr_value_t new_val,
            property_status_change_t property_status),
        cbdata_register_t data);
    attr_value_t (*notify_cpu_move_from)(
        conf_object_t *NOTNULL obj, uint64 node_id,
        void (*callback)(cbdata_call_t data, conf_object_t *obj,
            conf_object_t *cpu, attr_value_t node_path),
        cbdata_register_t data);
    attr_value_t (*notify_cpu_move_to)(
        conf_object_t *NOTNULL obj, uint64 node_id,
        void (*callback)(cbdata_call_t data, conf_object_t *obj,
            conf_object_t *cpu, attr_value_t node_path),
        cbdata_register_t data);
    attr_value_t (*notify_syscall)(
        conf_object_t *NOTNULL obj, uint64 node_id, bool recursive,
        void (*callback)(
            cbdata_call_t data, conf_object_t *obj,
            conf_object_t *cpu, uint64 node_id,
            const char *syscall),
        cbdata_register_t data);
    attr_value_t (*notify_callbacks_done)(
        conf_object_t *NOTNULL obj, uint64 node_id,
        void (*callback)(cbdata_call_t data, conf_object_t *obj),

```

```

        cbdata_register_t data);
attr_value_t (*notify_after_callbacks_done)(
    conf_object_t *NOTNULL obj, uint64 node_id,
    void (*callback)(cbdata_call_t data, conf_object_t *obj),
    cbdata_register_t data);
void (*cancel_notify)(
    conf_object_t *NOTNULL obj, uint64 callback_id);
bool (*set_property)(
    conf_object_t *NOTNULL obj, uint64 node_id,
    const char *key, attr_value_t value);
attr_value_t (*request)(conf_object_t *NOTNULL obj);
attr_value_t (*release)(conf_object_t *NOTNULL obj);
};
#define SOFTWARE_INTERFACE "software"

```

***root\_node\_id*** returns the ID of the root node. This ID never changes.

***get\_node*** returns a dictionary for a given node, containing all the node's properties; or nil if no such node exists.

***get\_subtree*** returns a dictionary mapping node IDs to node dictionaries; it contains the specified node and all its descendants. If the specified node does not exist, the return value is nil.

***get\_current\_nodes*** returns the current node path of the given processor. (That is, a list of node IDs of all nodes that are currently running on the processor, the first node being the root node, and every subsequent node a child of the node before it.) If the tracker is not tracking the given processor, the return value is nil.

***notify\_create*** and ***notify\_destroy*** register callbacks to be called when the node is created and destroyed, respectively. It is safe to read the node with ***get\_node*** from within the callback function.

***notify\_property\_change*** registers a callback that is triggered when the given property changes on the node (or any property, if *key* is NULL).

If recursive, the callback will be triggered for the given node's descendants as well as the node itself. All callbacks receive a *cpu* argument; it specifies the processor that caused the event, but may be NULL if the event was not caused by a processor.

***notify\_cpu\_move\_from*** and ***notify\_cpu\_move\_to*** register callbacks that are triggered when a processor moves from one node path to another—but only if either path lies in the subtree rooted at the given node.

***notify\_syscall*** registers a callback that will run whenever the software, represented by a node, makes a system call. The *obj* argument is the software tracker object. When registering the callback, the *node\_id* argument is the identifier for the node whose system calls should result in a call to the registered callback.

The *recursive* argument is used to tell the framework if system calls made by any of the children (or their children and so on) to the specified node should generate a callback or not. Trackers will associate the system call with the lowest entity of software, often referred to as a thread. If system calls generated by only one single thread is interesting then the *recursive* argument is not needed. However, if system calls for a process or the entire system

is interesting, then the *recursive* argument must be specified. This is required since the process node itself, nor the system node will be associated with any system call, only the thread will. The (*\*callback*) argument is the callback function that should be called upon a system call. The *data* argument will be passed back to the callback each time, and is not used by the framework. It can be left as `NULL` if the callback does not need it. The callback function must take five parameters. The first argument is the *data* that is the same as the *data* used when registering the callback function. The *obj* argument is the software tracker object. The processor that the tracker identified as the source for the system call will be given by the *cpu* argument. The identification of the node that represent the software that issued the system call will be given by *node\_id* argument. Finally the *syscall* argument represents the name of the system call, or if the tracker failed to convert the system call into a name it will contain the system call number prefixed with `sys_`, such as `sys_123`.

Since a single update to the node tree can result in several different callbacks being triggered, reading nodes with *get\_node* from a callback function may yield a result containing updates whose callbacks have not yet been run. (For example, if two nodes change their *name* attributes simultaneously, the final state of both nodes may be visible to both property change callbacks.) With *notify\_callbacks\_done*, you can register a callback that will run when all other callbacks pertaining to a particular change in the node tree are finished. Note that it is not possible to modify any node property at this stage.

With *notify\_after\_callbacks\_done*, you can register a callback that will run when all other callbacks pertaining to a particular change in the node tree and all *notify\_callbacks\_done* are finished. At this stage it is possible to modify node properties. Note though that other callbacks might have already run at this stage, which might have already changed node properties.

The functions that install callbacks return an integer ID, or `nil` on error. This ID can be passed to *cancel\_notify* in order to uninstall the callback.

*set\_property* sets a property on a node, and returns true if successful. Note that not all node properties are writeable.

*request* is used to register users that are interested in using the tracker framework and activates the tracker framework if it is not yet activated. It returns a text string upon errors that describes the error, otherwise `nil`.

*release* will decrease the number of users for each call and when no more users are interested in the tracker framework the trackers framework will be disabled. It returns a text string upon errors that describes the error, otherwise `nil`.

## 9.5 Using Context Objects

Contexts are Simics objects that represent virtual address spaces, such as the address space belonging to a process. In particular, virtual breakpoints (read, write, or execute breakpoints set on a virtual address) and symbol tables are associated with context objects.

At any moment, each processor object has a *current context*. By switching the current context, you switch between one set of virtual breakpoints and symbols, and another.

You have the option of manually switching the current context of the processors you are interested in. However, for the important case where you want to set the processors' contexts based on the node or nodes that are running on them, you can use the *set\_property* method

of the `software` interface to set the `context` property of one or more nodes to something other than `nil`. Once you have done so, the software tracker will make sure that the specified context is the current context of the processors that the node is running on.

In more detail: All nodes have a `context` property that is either `nil` or a context object. The software tracker sets the current context of its processors according to the following rule: of all the nodes with non-`nil context` that are running on a certain processor, pick the node farthest from the root and use its context. The three laws of nodes and processors (see section 9.3) guarantee that there is at most one such node.

## 9.6 System Call Notifications

System call notification is a way to register callbacks that should be called when the software makes a system call. The tracker will try to translate the system call number into the system call name, such as `fork` or `write`, and give the name to the callback function. If no translation was available, the name will be the system call number prefixed with `sys_`, like `sys_123`. For a list of the exact number of arguments that the callback will be called with, see the interface specification. Not all trackers support system call notifications, for those trackers this function will have no effect.

## 9.7 Node path patterns

A node in the node tree may be identified with its node ID. This is often not good enough, since it is hard to know the given node ID in advance; this makes scripting hard to accomplish in a satisfying way. Using **node path patterns** makes it possible to find nodes based on their properties instead of their node ID. A node path pattern is a matching rule that selects nodes based on one or more property values, and optionally the property values of their ancestors.

### 9.7.1 Introduction

This section will assume that the node tree looks like the one in figure 9.1.

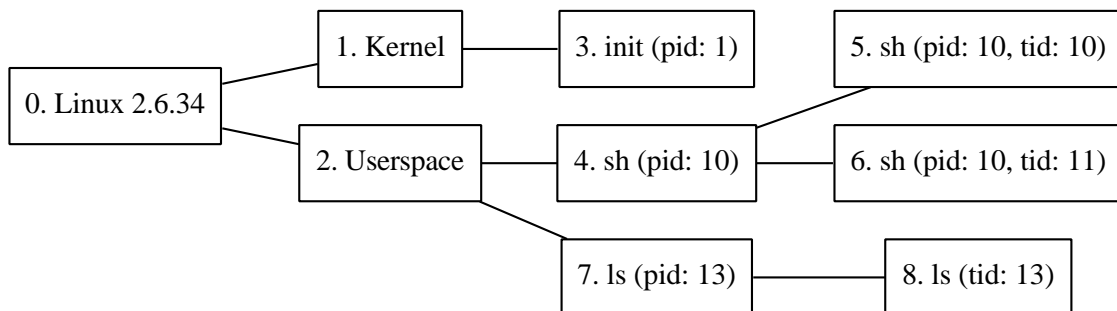


Figure 9.1: Example node tree

The simplest node path pattern is a single integer. This is treated as a node ID. Thus, “2” can be used to find the Userspace node:

```
simics> ebony.software.find 2
```

The name property will be used for matching if the given node specification only contains letters, digits, and underscore, and does not start with a digit. This gives an alternative way of locating the Userspace node in 9.1:

```
simics> ebony.software.find Userspace
```

However, if the name contain any non-alphanumeric characters, such as space or punctuation, the name needs to be enclosed in single quotes, and the name property needs to be named explicitly:

```
simics> ebony.software.find "name='Linux 2.6.34'"
```

A node path pattern can specify properties other than just the name of a node. For example, this one will find all nodes that belong to the sh process (the process node and all thread nodes) by specifying the matching pid:

```
simics> ebony.software.find "pid=10"
```

If a match is to be made based on multiple properties, they should be comma-separated. Given the node tree in figure 9.1, the following pattern would match just one node, the thread node with tid=10:

```
simics> ebony.software.find "pid=10,tid=10"
```

A slash (/) is used to separate rules for nodes at adjacent levels in the tree. For example, this pattern matches any node named sh that lies directly under a node named Userspace:

```
simics> ebony.software.find "Userspace/sh"
```

A slash at the beginning of the pattern anchors it to the root of the tree, so that the part after the first slash must match the root node, the part after the second slash must match any of its children, etc.

Node path patterns can contain three kinds of wildcards:

- One asterisk (\*) will match any *single* node. For example, /\*/\*/sh will match any node named sh that is a grandchild of the root node.
- Two asterisks (\*\*) will match a path of zero or more nodes in the tree. For example, /name='Linux 2.6.34'/\*\*/pid=10 matches nodes 4, 5, and 6 in the example tree.
- An asterisk in the expected value of a property matches zero or more characters. For example, \*s\* will match the ls, sh, and Userspace nodes; and pid=\* will match all nodes with a pid property.

### 9.7.2 Summary

This is a short summary of the available operators when creating a node path pattern.

- Integer value: this is treated as a node id
- String: if the string contains only alphanumeric characters or the `_` sign and does not begin with a digit, `str` will be the same as `name='str'`.
- Slash (`/`): separates rules for nodes at different levels. At the beginning of a pattern, it means that the root has to match the following rule.
- Asterisk (`*`): matches exactly one node, ignoring all node properties on that level.
- Double asterisk (`**`): Matches a path of zero or more nodes in the node tree.
- Equals sign (`=`): Specifies the required value of a node property.
- Comma (`,`): Separator when a rule restricts more than one node property.

### 9.7.3 Limitations

Node path patterns have some known limitations:

- Asterisk (`*`) is the only wildcard available for property matching. More advanced wildcards, such as `name=cat??` or `pid=5[0-1]`, are not supported.
- A rule may not end with `**/` or `/**`.
- It is not possible to start a pattern with `**/`; use `/**/` instead. (However, note that all patterns that do not already start with a slash are implicitly prefixed with `/**/`.)
- Just `**` is not a valid pattern. Use `*` in order to match all nodes.

## 9.8 Configuring the Software Tracker

The software tracker itself knows nothing about any particular operating system. It has a number of built-in modules, or *trackers*, that each knows about one operating system or hypervisor (see section 9.8.4), and someone must tell it which one to use, and with what settings.

There are three ways to configure the software tracker: the recommended **load-parameters** mentioned in section 9.8.1, by setting attributes when it is created, and by calling the `set_property` method of its `software` interface.

### 9.8.1 Configuration With Parameter Files

The easiest and recommended way to configure a tracker is to use one of the many detect commands that are provided. For example, such as **linux-autodetect-settings** command and **wr-hypervisor-detect-settings** command. All the detect commands will generate a parameters file that can be loaded directly by the **load-parameters** command. When the parameters file is loaded, the **enable-tracker** command will activate the newly configured tracker.

### 9.8.2 Configuration With Attributes

When you create a **software-tracker** object, two attributes provide the means of configuring it:

#### **processors**

A list containing the processor objects that run the software to be tracked. This list may not be modified once the software tracker has been instantiated.

#### **tracker**

A list with three items: the name of a tracker, the *settings* of the tracker, and the current *state* of the tracker. The attribute may also be nil, meaning that no tracker has been configured (this is the default value).

The type and meaning of the settings and state fields depend on the tracker; the settings are documented for each tracker in section 9.8.4. But in general, the *settings* are parameters that do not change, such as parameters specific to the exact version of the software being tracked, while the *state* is whatever internal state the tracker happens to have at the moment. Nil is always a legal initial state; you should use that when setting up the tracker. Non-nil states are only intended for checkpointing.

The software tracker is typically created as part of a component attached to the machine component; this automatically sets the *processors* attribute. See the end of e.g. `targets/mpc8641-simple/mpc8641-simple-system.include` for an example of how this is done.

The *tracker* attribute, which depends on the exact OS version, should be set by invoking the **load-parameters** command in the file that loads the OS image. See the end of e.g. `targets/mpc8641-simple/mpc8641-linux-setup-system.include` for an example of how this is done.

### 9.8.3 Configuration With Interface Calls

Once the **software-tracker** object is configured, you can inspect the nodes' `tracker` properties (using e.g. the `get_node` method of the `software` interface); the values are two-element lists of (tracker name, settings), or nil if no tracker is configured for the node.

Not all nodes have `tracker` properties, however; only those where a tracker is or can be attached. The most prominent example is the root node, but some trackers designate some nodes as points where a subordinate tracker can be attached. For example, a tracker for a hypervisor would allow subordinate trackers to be attached on the nodes representing

guests. Without a tracker attached, a guest node would be a leaf node; but with a tracker for the guest's operating system, the node would get children representing user-space, processes, and threads (or whatever abstractions the tracker in question provides).

Using the *set\_property* method of the `software` interface, you can write to the `tracker` property of any node that has it. This is intended to be used when you need to change trackers at run-time, such as when loading a new operating system. For initial configuration, use the **load-parameters** command as described in section 9.8.2.

## 9.8.4 List of Tracker Modules

### CPU Mode Tracker

The CPU mode tracker (`cpu_mode_tracker`) is the simplest tracker module; where other tracker modules have detailed knowledge about OS data structures, the CPU mode tracker looks only at whether each processor is currently in user or supervisor mode. It is intended to be used in cases where no specialized tracker is available.

Below the root node, you will find one node for user mode and one for supervisor mode; they are running on all processors that are currently in their respective modes. Below each of them, there is one node per processor in the system; these nodes are running on their processor when it is in the appropriate mode.

The CPU mode tracker defines just one node property in addition to the standard ones:

#### **cpu\_mode**

The mode of the processor or processors owned by this node; either `Sim_CPU_Mode_User` or `Sim_CPU_Mode_Supervisor`. It never changes.

This tracker has no settings; they should always be nil.

### Linux Tracker

The Linux tracker (`linux_tracker`) tracks processes and threads in Linux.

Once the OS has come far enough in the boot process for the first processes to start, the Linux tracker will create a tree like the one shown in figure 9.2 under the root node.

There are four kind of nodes:

#### **The kernel node**

(Node 1 in figure 9.2.) The kernel node is running on all processors that are currently executing kernel code.

#### **Kernel thread nodes**

(Nodes 3 and 4.) Immediately below the kernel node, there is one thread node for each kernel task. A new kernel thread node is created for each new kernel task. When the task is terminated, the node is destroyed.

#### **The user-space node**

(Node 2 in figure 9.2.) The user-space node is running on all processors that are currently executing user-space (i.e., non-kernel) code.



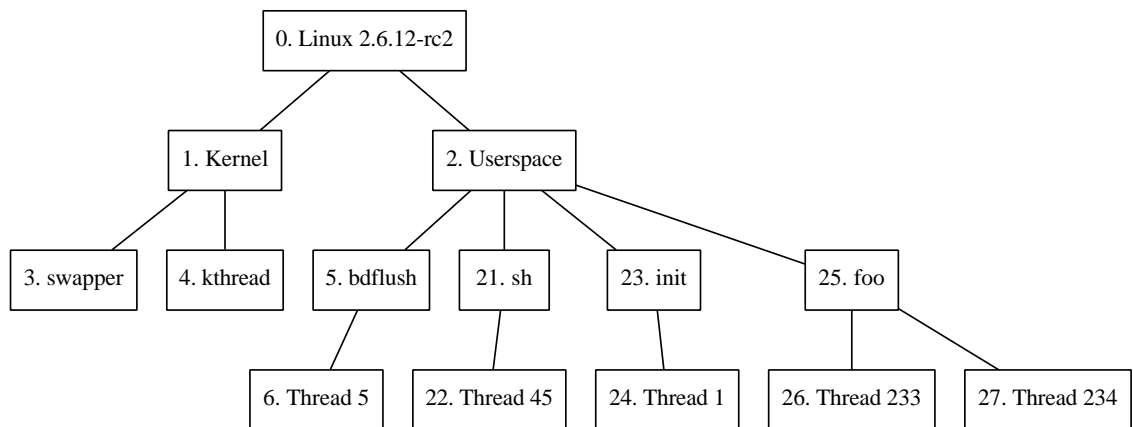


Figure 9.2: Node tree of a Linux tracker

**Process nodes**

(Nodes 5, 21, 23, and 25.) Immediately below the user-space node, there is one process node for each process in the system. A new process node is created for each new process, and when a process is terminated, its node is destroyed. Process nodes have the following extra properties in addition to the standard ones:

**binary**

A string containing the path to the binary that the process is running, or nil if the binary is not known.

**pid**

A list containing the pid of this process. This is the same as the task group ID.

**Thread nodes**

(Nodes 6, 22, 24, 26, and 27.) Immediately below each process node is one or more thread nodes representing the threads in the process. Just like the process nodes, these are created and destroyed on demand. Thread nodes have the following extra properties in addition to the standard ones:

**pid**

The integer pid (“process ID”) associated with the thread. This is usually the same for all threads in a process.

**tid**

The integer tid (“thread ID”) associated with the thread. This is unique in the whole OS instance.

**state**

The string “runnable” or the string “stopped”, depending on whether there is currently something preventing the thread from running.

The Linux tracker needs settings that tell it a bunch of stuff about the kernel it is tracking. Specifically, it needs a dictionary with the following keys:

**ts\_offsets**

A dictionary of offsets in the `task_struct`. The offsets of `state`, `next`, `prev`, `binfmt`, `pid`, `tgid`, `active_mm`, `mm`, and `comm` are always needed. `p_opptr`, `p_pptr`, `p_cptra`, `p_ysptr`, and `p_osptr` are needed for 2.4 kernels. `real_parent`, `parent`, `children_list_head`, and `sibling_list_head` are required for 2.6 kernels. The offset of `thread_struct` is needed on PowerPC 32.

**ts\_next\_relative**

True if the `prev` and `next` pointers point to the next pointer of their `task_struct`s; false if they point to the beginning of the structs.

**version\_string**

(optional) Kernel version string, to be displayed to the user. This is the “2.6.12-rc2” in figure 9.2.

**kernel\_base**

(32-bit only) Kernel start address; this is usually `0xc0000000`.

**ts\_indirect**

(x86 only) True if the stack top contains a pointer to the task struct, false if the task struct itself is there.

**stack\_size**

(x86 only) Kernel stack size in bytes; either 4096 or 8192.

**paca\_task\_struct**

(PowerPC 64 only) Offset of `thread_struct` pointer in the `paca` struct.

**Wind River Hypervisor**

When the Hypervisor has come far enough in the boot process, will the Hypervisor tracker create a tree like the one shown in figure 9.3 under the root node.

There are four kind of nodes:

**The Virtual Board parent node**

This node (1. in figure 9.3) is the parent node for all contexts that can host an operating system. That is, the context is a virtual board.

**Virtual board nodes**

These nodes (13. and 14. in figure 9.3) are place-holder nodes which represent guest operating systems. It is possible to attach an additional tracker to each of these nodes. The new tracker will only be active when the *guest* operating system is active. These nodes will contain shared properties between the hypervisor tracker and the guest operating system trackers. Virtual board nodes have the following extra properties in addition to the standard ones:

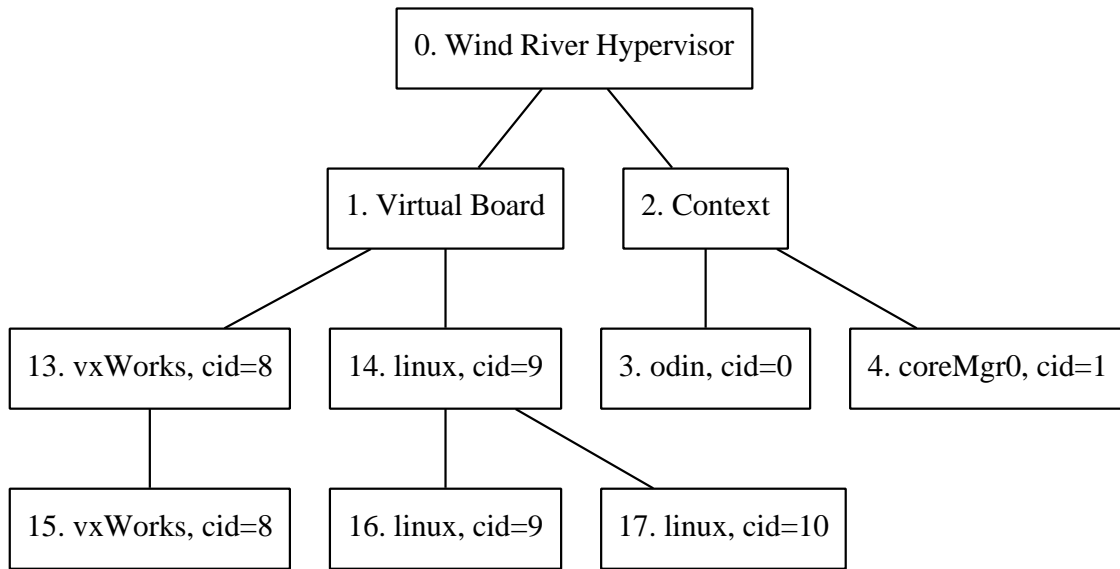


Figure 9.3: Node tree of a Wind River tracker

**cid**

The integer cid (“context ID”) associated with the context.

**tracker**

The configuration of the additional tracker that is attached to track this context.

**Virtual board context nodes**

These nodes, (15 to 17 in figure 9.3) represent hypervisor contexts for the processors the guest operating systems can execute on.

**The Context parent node**

This node (2. in figure 9.3) is the parent node to all context nodes that can not run an operating system, but are rather contexts that service the hypervisor itself.

**Context nodes**

There is one context node (3. and 4. in figure 9.3) for each context in the system, given that the context is not a virtual board. A new context node is created for each new context, and when the context is terminated, its node is destroyed. Context nodes have the following extra properties in addition to the standard ones:

**cid**

The integer cid (“context ID”) associated with the context.

**VxWorks Tracker**

The Wind River VxWorks tracker (`vxworks_tracker`) tracks tasks in VxWorks.

Once VxWorks is up and running the VxWorks tracker will create a tree like the one shown in figure 9.4.

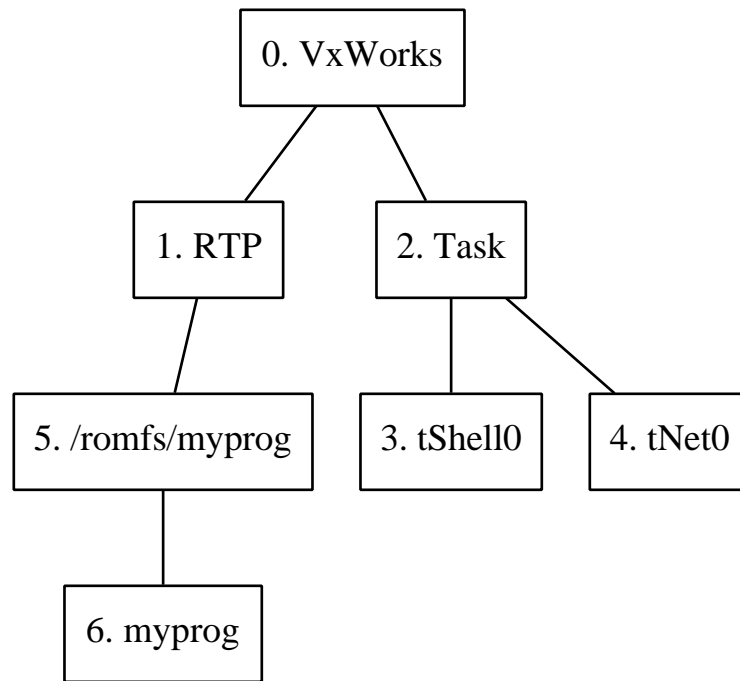


Figure 9.4: Node tree of a VxWorks tracker

There are three kind of nodes:

**The RTP parent node**

This node (1. in figure 9.4.) is used to group all Real Time Processes (RTP).

**The RTP program node**

This node (5. in figure 9.4.) acts like a placeholder for every thread belonging to the same program.

**The RTP thread node**

There is one RTP thread node (6. in figure 9.4.) for each RTP thread in the system. All threads that belong to the same RTP are grouped under the same node.

**The task parent node**

This node (2. in figure 9.4.) acts like a place holder for all kernel tasks.

**Task nodes**

There is one task node (3. and 4. in figure 9.4.) for each task in the system. A new task node is created for each new task, and when a task is terminated, its node is destroyed. Task nodes have the following extra properties in addition to the standard ones:

**tid**

The integer tid ("task ID") associated with the task.

## QNX Tracker

The QNX tracker (`qnx_tracker`) tracks user space processes in QNX.

Once QNX is up and running the QNX tracker will create a tree like the one shown in figure 9.5.

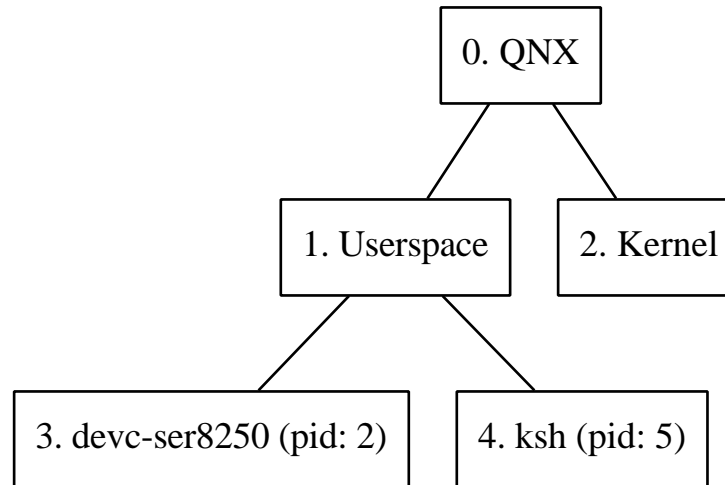


Figure 9.5: Node tree of a QNX tracker

There are three kind of nodes:

### The Userspace node

This node (1. in figure 9.5.) is used to group all userspace processes under one common node.

### The kernel node

This node (2. in figure 9.5.) The kernel node is active for all processors that are currently executing kernel code; it has no children, since the tracker does not keep track of separate tasks within the kernel.

### Process nodes

There is one process node (3. and 4. in figure 9.5.) for each process in the system immediately below the userspace node. A new process node is created for each new process, and when a process is terminated, its node is destroyed. Process nodes have the following extra properties in addition to the standard ones:

#### pid

The integer pid ("process ID") associated with the process.

## OSE Tracker

The OSE tracker (`ose_tracker`) tracks user space processes in OSE.

Once OSE is up and running the OSE tracker will create a tree like the one shown in figure 9.6.

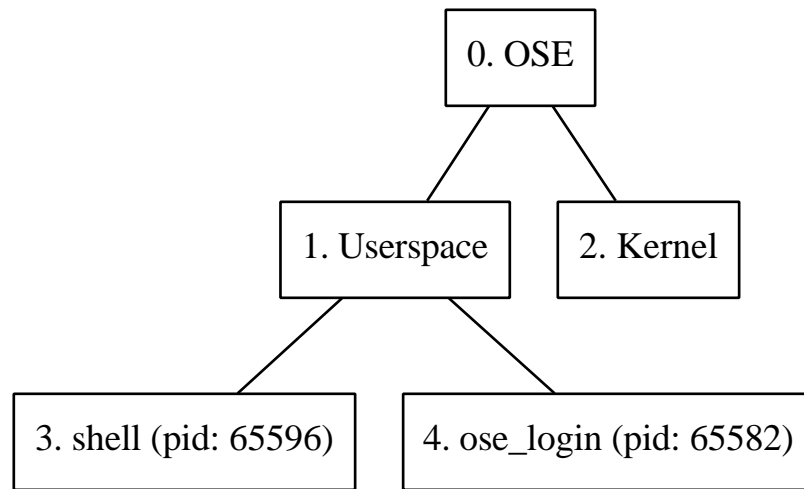


Figure 9.6: Node tree of a OSE tracker

There are three kind of nodes:

#### The Userspace node

This node (1. in figure 9.6.) is used to group all userspace processes under one common node.

#### The kernel node

This node (2. in figure 9.6.) The kernel node is active for all processors that are currently executing kernel code; it has no children, since the tracker does not keep track of separate tasks within the kernel.

#### Process nodes

There is one process node (3. and 4. in figure 9.6.) for each process in the system immediately below the userspace node. A new process node is created for each new process, and when a process is terminated, its node is destroyed. Process nodes have the following extra properties in addition to the standard ones:

##### pid

The integer pid ("process ID") associated with the process.

#### Partition Tracker

The Partition tracker (`partition_tracker`) can be used to statically group processors into different partitions.

The partition tracker allows grouping of processors that belongs to the same hierarchy. This can, for example, be useful when collecting coverage. If there is no tracker available for the target system, a simple partition tracker can place processors into different groups, allowing coverage to be collected on only a subset of them. The grouping of processors can also be useful for a system that has different operating systems running on the available

processors. The partition tracker does not support a subtracker to be attached to one of the nodes.

The node tree for a partition tracker is completely specified by its configuration. See figure 9.7 for an example configuration; as you can see, there can be any number of partitions, each with zero or more processors.

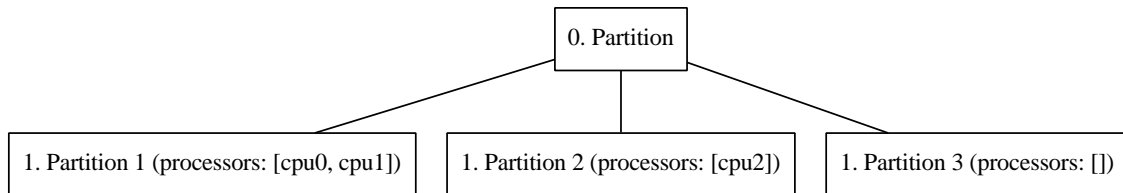


Figure 9.7: Node tree of a partition tracker

## 9.9 Using the Software Tracker in Scripts

This is an example script that uses the software tracker to track a process and counts all hardware exceptions that happen while the process is active. It assumes that the system component is **mpc8641d\_simple**

```

class exception_counter:
    "This class counts hardware exceptions for a specific process."

    def __init__(self, software_comp, process_name):
        self.tracker = software_comp.tracker
        self.notifiers = set()
        self.exc_haps = {}

        self.exceptions = {}          # The result

        # The node names will be truncated to 15 characters, since
        # they use the Linux task 'comm' field. So we only match the
        # first 15 characters of the requested process name.
        process_name = process_name[:15]

        self.sw = self.tracker.iface.software

        # Install a callback on node creation
        self.notifiers.add(
            self.sw.notify_create(self.sw.root_node_id(), True,
                                  self.create_cb, process_name))

        # Install a callback on changes to the 'name' property in any node, in
        # case the program switches name after the node was created

```

```

self.notifiers.add(
    self.sw.notify_property_change(self.sw.root_node_id(), "name", True,
                                   self.name_cb, process_name))

print ("Will count exceptions for the next process called %s"
       % process_name)

def is_process(self, node_id):
    # This will only work for the Linux tracker. It uses the fact that a
    # process node contains the process id, but not the thread id
    props = self.sw.get_node(node_id)
    return 'pid' in props and not 'tid' in props

def create_cb(self, process_name, tracker, curcpu, node_id):
    # There can be other nodes than the process node with a
    # matching name, for example thread nodes. Verify both name
    # and that it is a process.
    if (self.sw.get_node(node_id)['name'] == process_name
        and self.is_process(node_id)):
        self.process_found(node_id)

def name_cb(self, process_name, tracker, curcpu, node_id,
            key, old_val, new_val, status):
    # There can be other nodes than the process node with a
    # matching name, for example thread nodes. Verify both name
    # and that it is a process.
    if new_val == process_name and self.is_process(node_id):
        self.process_found(node_id)

def process_found(self, node_id):
    # Remove the callbacks for node creation and name changes
    for n in self.notifiers:
        self.tracker.iface.software.cancel_notify(n)

    # Install callbacks when processors enter and leave this
    # process node.
    self.notifiers.add(
        self.sw.notify_cpu_move_to(node_id, self.move_to_cb, None))
    self.notifiers.add(
        self.sw.notify_cpu_move_from(node_id, self.move_from_cb, None))

    # Install a callback when the process finishes
    self.notifiers.add(
        self.sw.notify_destroy(node_id, False, self.destroy_cb, None))

    # For each CPU already executing in this node, make sure
    # to enable counting.
    props = self.tracker.iface.software.get_node(node_id)
    for cpu in props['processors']:
        self.enable_counting(cpu)

```



```

def enable_counting(self, cpu):
    # Install a hap callback for the exception hap
    self.exc_haps[cpu] = SIM_hap_add_callback_obj("Core_Exception", cpu, 0,
                                                self.exception_cb, None)

def disable_counting(self, cpu):
    SIM_hap_delete_callback_id("Core_Exception", self.exc_haps[cpu])

def move_to_cb(self, data, tracker, cpu, node_path):
    self.enable_counting(cpu)

def move_from_cb(self, data, tracker, cpu, node_path):
    self.disable_counting(cpu)

def destroy_cb(self, data, tracker, cpu, node_id):
    print "The process finished"
    for exc in sorted(self.exceptions.keys()):
        print "%5d %-30s: %8d" % (exc, cpu.iface.exception.get_name(exc),
                                self.exceptions[exc])
    for n in self.notifiers:
        self.tracker.iface.software.cancel_notify(n)

def exception_cb(self, data, cpu, exception):
    if exception in self.exceptions:
        self.exceptions[exception] += 1
    else:
        self.exceptions[exception] = 1

counter = exception_counter(conf.mpc8641d_simple.software, "ls")

```

---

**Note:** Remember that this will only work if the tracker is enabled. It can be enabled with the **mpc8641d\_simple.software.enable-tracker** command, and it will also be enabled automatically by most tracker-related commands.

---

The script works by listening to notifications of processors moving between different tracker nodes, which means that it could be entering or leaving the process context we are profiling.

## Chapter 10

# Graphical Timeline Example

This chapter will give a short introduction on how the Graphical Timeline view can be used. It is part of Simics Eclipse and is meant to give a fair representation of how processes are scheduled over simulated time. It can be used to analyze the system load and display areas of interest for optimizations. We will start off by analyzing the boot of the “PPC Firststeps” machine and then continue to analyze a simple userspace program. The Graphical Timeline view requires OS awareness availability. The example programs can be found in the “PPC Firststeps” package target directory.

### 10.1 Analysis of the Firststeps Linux boot

In this section we will start by analyzing the Linux boot on the two core MPC Firststeps machine. Then a brief discussion on what would be potential ways of optimizing the system, but also an example of what Graphical Timeline would not be good for.

Load the `mpc8641-simple/firststeps.simics` script in Eclipse. We are going to consider the system fully booted when we get access to the prompt. You can set a breakpoint that will stop the simulation with the following command:

```
simics> mpc8641d_simple.console0.con.break "~ #"
```

Make sure to open the Graphical Timeline view before you start the simulation: open the Simics perspective under “Window→Open Perspective”; then open the Graphical Timeline view under “Window→Show View”.

In figure 10.1 the Graphical Timeline shows the entire boot, which takes approximately 2 seconds. To see this, make sure to pan out as far as possible using the zoom icon at the bottom left. The first 200ms does not contain any information about the system. This is due to the fact that time spent in BIOS, boot loader and early Linux boot will not be tracked by the Linux tracker.

After some userspace activity between roughly 200ms and 800ms we can see that the entire system is idling until the boot is completed about 1 second later.

The important lesson from this simple example of the Graphical Timeline view is to see how it can aid in decisions on where to put development resources and hardware requirements. There is no need to buy 4 cores, if the software never uses more than one core.

Maybe it is possible to shorten the time where the system is idle, probably waiting for a timeout. The Graphical Timeline view will also make it easier to get a rough estimate on how much time that can possibly be saved by parallelizing execution.

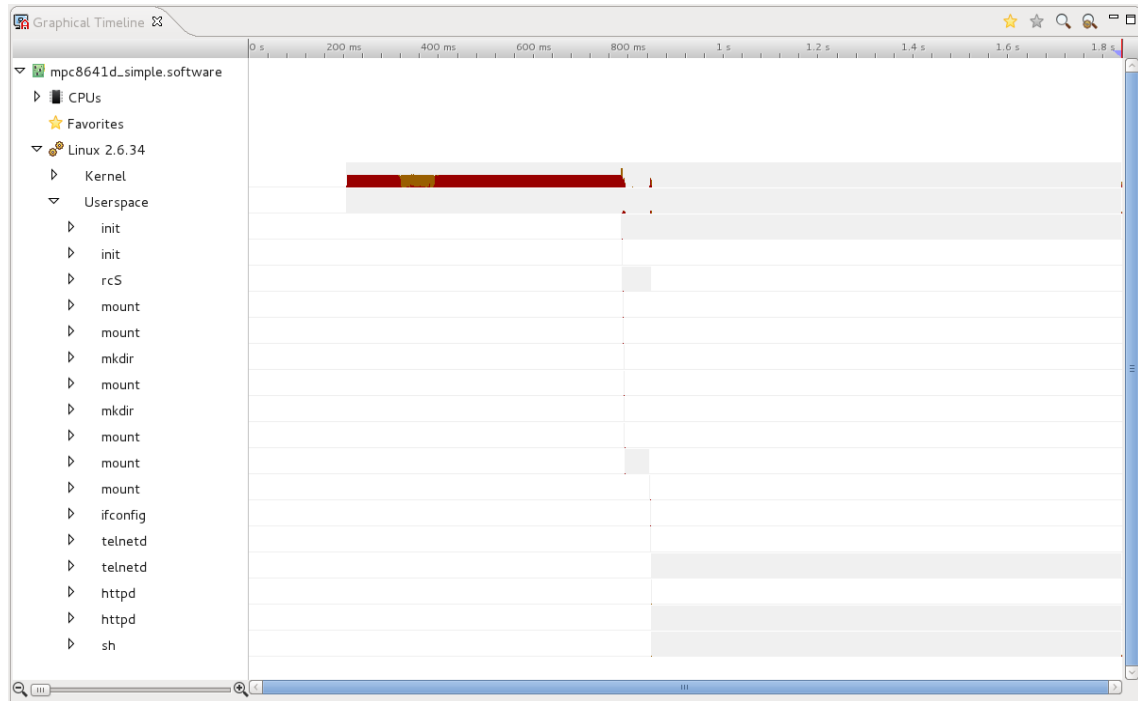


Figure 10.1: Graphical Timeline view representing the boot of Firststeps machine on PPC

## 10.2 Analyzing of a simple userspace program

In this section we will use the Graphical Timeline view to analyze a basic userspace program, which calculates the Fibonacci series. The program is very basic when it comes to error handling, optimization etc. So it should simply be seen as a way to demonstrate how the Graphical Timeline view can be used, and how to interpret the resulting information. The source code and the binaries are provided by the Firststeps package and located in the target directory.

This example is based upon the mpc8641 firststeps system. The first thing we have to do is to copy the binaries into the target system, this can be done by mounting the host system over Simics FS.

Make sure to open the Graphical Timeline view in Eclipse, if it is not already opened, before starting the program. The program will calculate the Fibonacci series for each input argument.

```
~ # /worker 36 37 41 35
Fibonacci for 36 is 14930352
```

## 10.2. Analyzing of a simple userspace program

```
Fibonacci for 37 is 24157817
Fibonacci for 35 is 9227465
Fibonacci for 41 is 165580141
```

After the *worker* program has completed you can pause the simulation. This makes it slightly easier to analyze the system, since there will not be any more data that may affect the Graphical Timeline view. Expand the Graphical Timeline view and select the *worker* process under the Userspace node. Now click on the “Add Favorite” button at the top right corner of the view. This will make that process a favorite. That is, the process will be placed under the favorite node, but also highlighted with its own color. They can later be removed by first selecting the favorite and then press the “Remove Favorite” button. Now, go up to the top of the view, expand the Favorites tab and the *worker* process. This will show both the process and the threads. Select the process node and click on the “Zoom in on existence” button at the top right corner of the view. This will zoom in on the selected program, so that only its time of existence is shown.

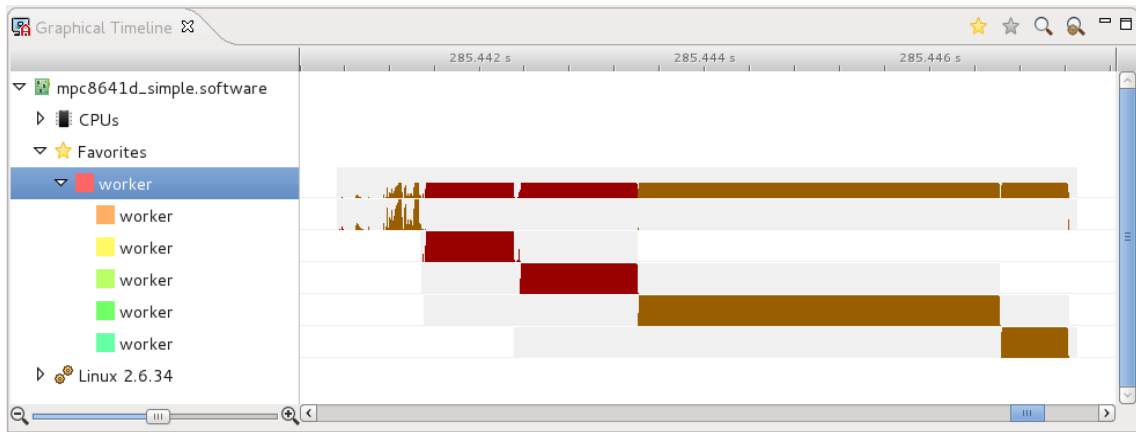


Figure 10.2: Graphical Timeline view representing of unoptimized program

The zoomed in view of *worker*, as seen in figure 10.2 shows its activity on the different cores on the system. Here one of the cores is colored red, and the other orange. This can also be seen under the *CPUs* node (collapsed). The *worker* process node is the combined activity of its threads. It is clear that the program does not utilize all processors in parallel.

In this case this was done on purpose, but for a normal program the Graphical Timeline view can help to verify that the program uses as many threads as expected, and that they actually do run in parallel. Another thing to notice here is that the program first starts to execute on processor 1, then processor 0 and back to processor 1. This gives visibility into what the scheduler is doing, that would have been harder to see without the Graphical Timeline view.

Now, let's study the slightly optimized version of the same program, using the same input parameters.

```
~ # /worker-opt 36 37 41 35
```

## 10.2. Analyzing of a simple userspace program

```
Fibonacci for 35 is 9227465  
Fibonacci for 36 is 14930352  
Fibonacci for 37 is 24157817  
Fibonacci for 41 is 165580141
```

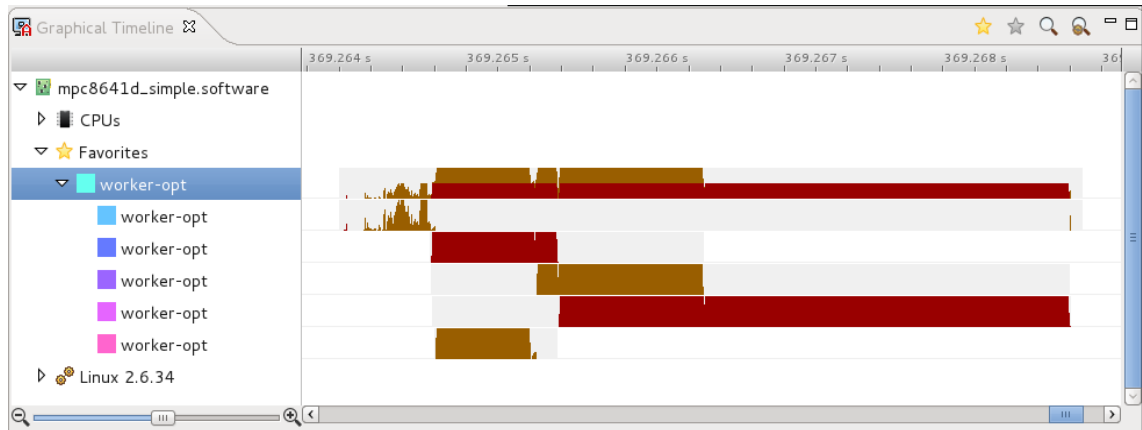


Figure 10.3: Graphical Timeline view representing the slightly optimized program

Figure 10.3 shows the *worker-opt* program in the Graphical Timeline. Here we can see that the program now utilize both of the cores at the same time. This has shortened the execution time, notice the scale difference. However, we can still see that we do not use the system to its full capacity. This is a good example of how the Graphical Timeline view at a glance displays potential performance issue.

# Index

## B

- binary
  - software tracker node property, [89](#)
- branch recorder, [53](#)
- Breakpoints
  - Eclipse
    - context creation, [32](#)
    - context destruction, [32](#)
    - function, [31](#)
    - icons, [35](#)
    - ignore count, [33](#)
    - line breakpoints, [31](#)
    - manipulating breakpoints, [34](#)
    - physical address, [33](#)
    - scope, [35](#)
    - status, [34](#)
    - watchpoints, [31](#)

## C

- caches
  - simulation, [59](#)
  - workload positioning, [65](#)
- cancel\_notify
  - software interface method, [83](#)
- children
  - software tracker node property, [79](#)
- cid
  - software tracker node property, [90](#), [91](#)
- code coverage, [48](#)
- code-coverage, [49](#)
- context, [14](#), [46](#)
  - software tracker node property, [79](#)
  - switching, [83](#)
- coverage analysis, [48](#)
- CPU mode tracker, [88](#)
- cpu\_mode
  - software tracker node property, [88](#)

cpu\_mode\_tracker, [88](#)

## D

- debugger, [22](#)
- debugging, [9](#), [22](#), [46](#), [76](#)

## E

- Eclipse, [22](#)
  - Breakpoints
    - context creation, [32](#)
    - context destruction, [32](#)
    - function, [31](#)
    - icons, [35](#)
    - ignore count, [33](#)
    - line breakpoints, [31](#)
    - manipulating breakpoints, [34](#)
    - physical address, [33](#)
    - scope, [35](#)
    - status, [34](#)
    - watchpoints, [31](#)
  - debug configuration, [24](#)
  - debug view, [23](#)
  - Symbol Browser, [25](#)
  - symbols, [25](#)
    - adding file, [26](#)
    - limitations, [28](#)
    - map section, [27](#)
    - re-map segment, [27](#)
- extra\_id
  - software tracker node property, [79](#)

## G

- g-cache, [59](#)
  - workload positioning, [65](#)
- generic-cache, [59](#)
- get\_current\_nodes
  - software interface method, [82](#)
- get\_node

- software interface method, 82, 87
- get\_subtree
  - software interface method, 82

**H**

- hypervisor, 9, 76

**I**

- instruction fetches, 65
- instruction profiling, 48, 53

**L**

- Limitations, 44
- Linux tracker, 88
- linux\_tracker, 88

**M**

- memory\_space
  - software tracker node property, 79
- multiprocessor
  - software tracker node property, 79

**N**

- name
  - software tracker node property, 79
- node ID
  - software tracker, 78
- node path pattern, 84
- node property
  - software tracker, 78
- notify\_after\_callbacks\_done
  - software interface method, 83
- notify\_callbacks\_done
  - software interface method, 83
- notify\_cpu\_move\_from
  - software interface method, 82
- notify\_cpu\_move\_to
  - software interface method, 82
- notify\_create
  - software interface method, 82
- notify\_destroy
  - software interface method, 82
- notify\_property\_change
  - software interface method, 82
- notify\_syscall
  - software interface method, 82

**O**

- operating system, 9, 76
- OS awareness, 9
- OS awareness details, 76
- OSE, 44
- OSE tracker, 93
- ose\_tracker, 93

**P**

- parent
  - software tracker node property, 79
- Partition tracker, 94
- partition\_tracker, 94
- pid
  - software tracker node property, 89
  - software tracker node property, 89, 93, 94
- processors
  - software tracker attribute, 87
  - software tracker node property, 79
- profiling, 53

**Q**

- QNX, 44
- QNX tracker, 93
- qnx\_tracker, 93

**R**

- release
  - software interface method, 83
- request
  - software interface method, 83
- root\_node\_id
  - software interface method, 82

**S**

- set\_property
  - software interface method, 79, 80
  - software interface method, 83, 86, 88
- Simics Analyzer, 7
- software domain, 20
- software tracker
  - configuration, 87
  - configuring, 86
  - initial configuration, 87
  - runtime configuration, 87

software\_interface\_t, [80](#)

stall mode, [48](#), [53](#)

state

    software tracker node property, [89](#)

syntable, [46](#)

## T

target software, [9](#), [48](#)

tid

    software tracker node property, [89](#)

    software tracker node property, [92](#)

tracker, [76](#)

    software tracker attribute, [87](#)

    software tracker node property, [80](#), [87](#),  
    [91](#)

tracker modules, [77](#)

## V

vxworks\_tracker, [91](#)

## W

Wind River VxWorks tracker, [91](#)