Flexus Getting Started Guide

Flexus 3.0.0

11/29/2007

Guide author: Evangelos Vlachos

SimFlex Project Team

Eric Chung, Michael Ferdman, Brian Gold, Nikos Hardavellas, Jangwoo Kim,
Ippokratis Pandis, Minglong Shao, Jared Smolens, Stephen Somogyi,
Evangelos Vlachos, Thomas Wenisch, Roland Wunderlich,
Anastassia Ailamaki, Babak Falsafi and James C. Hoe

Computer Architecture Lab at
Carnegie Mellon

Legal Notices

Introduction

Computer architects have long relied on software simulation to evaluate the functionality and performance of architectural innovations. Unfortunately, there a number of hurdles in sight that make simulation a design bottleneck. First, modern cycle-accurate simulators are several orders of magnitude slower than real hardware. Second, the growing levels of integration continue to result in commensurate increases in computer system size and complexity. Third, benchmarking commercial server software requires full-system simulation including the simulation of peripheral devices and OS code. Finally, because conventional simulators are optimized for speed, they are often not component-based and as such are inflexible and difficult to modify.

The SimFlex project has developed simulation tools and measurement methodology to enable fast, accurate, and flexible performance evaluation of uni- and multiprocessor systems running unmodified commercial applications. SimFlex combines rigorous statistical sampling, to choose application subsets for measurement, with reusable checkpoints of system state, to enable rapid simulation of the selected sample. Together, these techniques enable ~10,000 times reduction in measurement relative to cycle-accurate simulation without sampling and up to 1000-way simulation parallelism over a cluster of simulation hosts.

Flexus

Flexus is a family of component-based C++ computer architecture simulators that build on Virtutech Simics's Micro-Architecture Interface (MAI) to enable full-system timing-accurate simulation of uni- and multiprocessor systems running unmodified commercial applications and operating systems.

Flexus encompasses both a simulation infrastructure and default simulation models. A simulator is composed of individual modules that are hooked together during compilation. A module is often the equivalent of a single hardware structure—for example, a branch predictor or a cache. A key strength of Flexus is its isolation of components: one implementation of a particular module can be swapped for a different implementation without requiring changes to any other modules. This flexibility also allows a particular simulator to be tailored to the needs of a specific research hypothesis. If memory system performance is being evaluated, a simple bandwidth-based processor pipeline might be sufficient. Conversely, a study that closely examines microarchitecture could use a simple memory system model. The Flexus core provides services, such as scheduling and statistics, which are common and useful to all simulators.

Virtutech Simics enables full-system simulation. Simics is a functional simulator that allows unmodified commercial operating systems and applications to boot and run. Flexus can hook into Simics and see the instruction stream that a real system would execute. Flexus can also control Simics's timing, so as to model out-of-order effects and speculative techniques.

Flexus is designed to support the simulation sampling and checkpointing methodologies developed by the SimFlex research project. The keys to this support are *flex-points*, checkpoints that store the snapshots of the state of Flexus components alongside Simics checkpoints of programmer-visible state, and `stat-manager`, a tool for processing Flexus statistics output to create reports and compute statistical confidence in results.

Flexus's component-based design enables easy creation of several components that all model the same hardware at various levels of timing fidelity. These components all share the same format for storing state in flex-points. In this way, a simple, fast simulator can rapidly construct a flex-point library, which can then be measured using a more detailed detailed simulator. For example, the Flexus `TraceFlex` simulator creates flex-points that can be used for detailed out-of-order timing simulation with `UniFlex.OoO`.

The Flexus Getting Started Guide assumes that readers have some familiarity with the SimFlex experiment methodology and the SMARTS and Live-points simulation sampling techniques. If you have not already done so, we strongly recommend reviewing the SimFlex tutorial slides, and reading the following publications (all available on the SimFlex web site at http://www.ece.cmu.edu/~simflex):

- T. F. Wenisch, R. E. Wunderlich, M. Ferdman, A. Ailamaki, B. Falsafi, J. C. Hoe, "SimFlex: Statistical Sampling of Computer Architecture Simulation," *IEEE Micro Special Issue on Computer Architecture Simulation*, July/August 2006.

- T. F. Wenisch, R. E. Wunderlich, B. Falsafi, J. C. Hoe, “Simulation Sampling with Live-points,” *Proceedings of the International Symposium on Performance Analysis of Software and Systems (ISPASS)*, March 2006.
- R. E. Wunderlich, T. F. Wenisch, B. Falsafi, J. C. Hoe, “SMARTS: Accelerating Microarchitecture Simulation via Rigorous Statistical Sampling,” *Proceedings of the 30th International Symposium on Computer Architecture (ISCA)*, June 2003.

This release of Flexus includes several example Flexus simulation models:

- `TraceFlex` A non-timing simulator used to study cache miss rates and to construct flex-points for `UniFlex`(`.OoO`) and `DSMFlex`(`.OoO`)
- `TraceCMPFlex` A non-timing simulator used to study CMP cache miss rates and to construct flex-points for `CMPFlex`(`.OoO`)
- `CMPFlex` In-order chip multiprocessor, private L1s, shared L2, based loosely on Compaq Piranha
- `CMPFlex.OoO` `CMPFlex` with an out-of-order processor core
- `UniFlex` In-order uniprocessor with split L1 I/D and unified L2
- `UniFlex.OoO` `UniFlex` with an out-of-order processor core
- `DSMFlex` A directory-based distributed-shared-memory multiprocessor, where each node is similar to the `UniFlex` model
- `DSMFlex.OoO` `DSMFlex` with an out-of-order processor core

System Requirements

Operating System

Flexus 3.0.0 runs only under Linux. It has been tested on the following Linux releases:

- SuSE Linux 10.1
- SuSE Linux 8.1
- Fedora Core 8
- Fedora Core 7
- Fedora Core 5

External Software

Flexus 3.0.0 depends on a number of external tools and software libraries:

- GNU GCC 4.1
- Boost C++ library 1.33.1
- Virtutech Simics 3.0.X
- GNU make 3.79.1 or newer

GNU GCC 4.x

Flexus uses several advanced features of C++ and requires GCC 4.1 or higher. It will compile with the pre-installed version of GCC on Fedora Core 5, Fedora Core 7 and Fedora Core 8. On other Linux releases, it may be necessary to rebuild GCC from source. Flexus has been tested with GCC 4.1.0 (Fedora Core 5), GCC 4.1.2 (Fedora Core 7 and 8) and GCC 4.1.0 (built from GNU sources).

When building GCC, we suggest using the --prefix and --program-suffix options of GCC's "configure" script to install GCC 4.1.0 with alternate file names and to an alternate installation directory (rather than install over your system's current gcc installation). We also recommend configuring gcc with --disablethreads, as this will improve the performance of Flexus by nearly 30%. Here is the sequence of commands we used to build gcc:

(assumes gcc 4.1.0 is untarred in ./gcc-4.1.0)

```
mkdir gcc-4.1.0-build
cd gcc-4.1.0-build
../gcc-4.1.0/configure --with-prefix=/home/gcc-4.1.0 --with-suffix=-4.1.0
--disable-threads --enable-languages=c,c++
make profiledbootstrap
make install
```

Complete documentation on GCC is available at http://gcc.gnu.org/. GCC 4.1 can be downloaded from any of the mirror sites listed at http://gcc.gnu.org/mirrors.html. Instructions on installing and building GCC are provided at http://gcc.gnu.org/install/. We provide current links to these sites on the SimFlex web page (http://www.ece.cmu.edu/~simflex/). You will need to specify the path and executable name for GCC 4.1 when setting up Flexus.

Boost C++ 1.33.1

The Boost C++ libraries are a set of freely redistributable source code libraries for C++, and are used extensively throughout Flexus. Flexus will not compile with Boost versions other than 1.33.1.

The Boost libraries are documented at http://www.boost.org/. We provide a link to the Boost 1.33.1 download site on the SimFlex web page (http://www.ece.cmu.edu/~simflex/).

Once you have downloaded Boost, unpack the distribution to a desired location (e.g., /home/boost). You will need to specify the path where Boost is installed when setting up Flexus. It is not necessary to compile any parts of Boost–the Flexus makefiles will do this for you.

Virtutech Simics

Flexus runs as a loadable module in Simics. We have tested this Flexus 3.0.0 release with Simics 3.0.21, 3.0.22, 3.0.27 and 3.0.30 (latest as of the time writing this document). The scripts included with this release will not work with versions of Simics older than Simics 3.x. Simics 3.x versions older than 3.0.21 have not been tested, but will likely work without modifications to Flexus.

Virtutech provides complete documentation on installing Simics. You should verify that you can run Simics successfully before proceeding with Flexus.

Please contact Virtutech (http://www.simics.net/) for more information on obtaining and installing Simics. Personal licenses are available to academic users at no charge (https://www.simics.net/evaluation/academic.php).

Flexus Distribution

The latest Flexus distribution is always available at http://www.ece.cmu.edu/~simflex/.

GLIBC compatibility

Because Flexus is built with a new version of GCC, it requires the latest GLIBC runtime libraries to function properly. Simics is built with an older version of gcc (3.2.2). Hence, Virtutech ships the GLIBC runtime libraries from gcc 3.2.2 with Simics. These older GLIBC libraries are not compatible with Flexus. As part of the Flexus installation process, the Flexus makefiles replace the GLIBC libraries in the Simics distribution with newer libraries. On many Linux distributions (e.g., SuSE Linux 10.1, SuSE Linux 8.1, Fedora Core 8, Fedora Core 7 and Fedora Core 5), both Flexus and Simics will work without difficulty using the newer libraries.

Building Flexus

Preparing the Build Environment

The following procedure will help you set up the Flexus 3.0.0 code and make system to build in your environment.

1. Install GCC, Boost 1.33.1, and Simics 3.0.xx as described above.
2. Unpack the Flexus source distribution. Throughout this documentation, we refer to the directory where you unpack the Flexus source as `FLEXUS_ROOT`.
3. Modify the paths in `FLEXUS_ROOT/makefile.defs`. The Flexus makefiles need to know the locations where GCC, Boost, and Simics are installed. You specify these in `makefile.defs`. You must set the following four make variables:

   `GCC_PATH` path where GCC 4.0.0 or 4.1.0 is installed. This is the root directory of the GCC installation, which you specified with the `--with-prefix` option when configuring GCC. If using the system's built-in GCC, set this to `/usr`.

   `GCC_BINARY` The name of the g++ executable. If you build GCC using our recommended options, it will be `g++-4.1.0`. The system's built-in binary is `g++`.

   `BOOST_PATH` Path where the Boost 1.33.1 distribution has been unpacked. This directory contains boost, libs, and doc subdirectories, among others.

   `SIMICS_PATH` Path where Simics has been installed. This directory contains x86-linux, scripts, and config subdirectories, among others.

   The Flexus makefiles will verify these paths and check the versions of each external dependency every time the makefiles are used. The version checks can be disabled if you want to use Flexus with unsupported versions of the external tools (see `makefile.checksetup` for details).

4. To install the GLIBC runtime libraries into the Simics distribution, enter the `FLEXUS_ROOT` directory and type:

   ```
   make install
   ```

   Flexus will install GLIBC runtime libraries from GCC 4 into the Simics distribution. See the preceeding discussion of library compatibility for details. This command replaces the GLIBC runtime libraries in `SIMICS_PATH/x86-linux/sys/lib`, and backs the original libraries up in `SIMICS_PATH/x86-linux/sys/lib/`. Running `make uninstall` will cause Flexus to restore the original libraries.

Flexus is now fully installed and ready to build simulators.

Compiling Flexus

Flexus targets are built by issuing the command:

```
make <target>
```

where <target> consists of:

```
<component or simulator name>[-<debug level>][-<subtarget>][-<architecture>]
```

If you run make without any arguments, Flexus will list the components and simulators it knows how to build. The Flexus make system automatically determines the set of simulators and components that can be built by examining the contents of the `components` and `simulators` subdirectories. Thus, any new components or simulators that are added will automatically be recognized by the makefiles. Each simulator includes a makefile called `FLEXUS_ROOT/simulators/<simulator name>/Makefile.<simulator name>` that indicates the set of components required by that simulator. All of a simulators' dependencies are built before the simulator.

The Flexus debug system allows debug statements to be included or excluded from compilation from the make command line. Each debug statement has an associated severity level. The optional `-<debug level>` portion of the make target controls the minimum debug severity level included in the build. Debug statements with lower severity are excluded from compilation through preprocessor macros. Debug severity levels are documented in the SimFlex tutorial.

The optional `-<subtarget>` portion of the make target allows special operations or variants of a make target to be built. All make targets support a `-clean` subtarget, which deletes all intermediate object files that may exist for that target, and a `-clobber` subtarget, which additionally removes the output binaries and dependency files created by a target.

In addition to components and simulators, the Flexus make system supports a number of special targets:

- **`all`** Builds, cleans, or clobbers all simulators and components
- **`install`** Install compatible GLIBC into Simics
- **`uninstall`** Restore original GLIBC libraries
- **`stat-manager`** Build the stat-manager tools

The optional `-<architecture>` portion defines the architecture that Flexus will be targeted. If not defined, Flexus will be compiled for Sparc V9 architectures. X86 targets can be defined by the "-x86" flag.

Here are some example make command lines:

`make all-clobber`

Completely removes all intermediate files and binaries from the Flexus source tree. If you experience unexpected "Simics getting shaky errors" when running with Flexus, we suggest clobbering and rebuilding Flexus to ensure that your build is consistent.

`make UniFlex.OoO-iface`

Builds the UniFlex.OoO simulator and all its dependencies with iface severity debug statements enabled.

`make UniFlex-x86-vverb`

Builds the in-order UniFlex simulator and all its dependencies with vverb severity debug statements enabled, for x86 architectures.

Running Experiments in Flexus

The following sections walk through an example timing experiment with Flexus. Over the course of the following sections, you will perform all the steps necessary to prepare a flex-point library and create a report of the results of that experiment, including confidence interval calculations. The example experiment makes use of a variety of scripts included in the Flexus distribution to automate the steps of the process. We suggest using these scripts as a starting point for using your own applications with Simics and Flexus. To complete this experiment, you will require approximately 2GB of free diskspace. Any warnings that you might encounter during this process are listed in the Appendix.

The test application is a simple parallelized image processing filter that applies a blur filter to a ppm image. The filter is repeatedly applied to the image, to emulate iterations as typically found in parallel application benchmarks (e.g., SPLASH). Once the simulations are complete, you will estimate the IPC (instructions per cycle) of the application with an associated confidence interval, and construct a breakdown of execution stalls. (Optionally, you may construct an Excel graph depicting the execution time breakdown.)

The complete procedure to prepare and run the sample experiment involves the following steps.

1. Install and compile Flexus.
2. Install the simulated operating system image in Simics
3. Create a Simics checkpoint with the simulated operating system booted and running the `flexus-test-app`
4. Create a flex-point library for detailed simulation of the second iteration of `flexus-test-app`
5. Run out-of-order simulations of the flex-point library
6. Aggregate the statistics databases created by Flexus and generate reports

The subsections below detail each step.

Installing and Building Flexus

Follow the steps in the previous section to obtain the necessary software, install, and build Flexus. To complete the example experiment, you must build the following targets: (see also Appendix:A1)

SPARC simulated architectures

```
make TraceFlex
make UniFlex.OoO
make stat-manager
```

x86 simulated architectures

```
make TraceFlex-x86
make UniFlex-x86
make stat-manager
```

OS installation

Before you can run Flexus or any other applications, you must construct a Simics disk image of an operating system.

Installing Solaris

Solaris ISO images are freely available from Sun. The flexus-test-app has been tested with Solaris 10. As of this writing, Solaris 10 CD images can be downloaded from http://www.sun.com/software/solaris/get.jsp. Simics includes scripts to automate the Solaris installation. Detailed instructions on how to install Solaris under Simics are in Chapter 6 of the Simics/Serengeti Target Guide, pages 56-57. Simics 3.0.22 includes a Serengeti system configuration called “abisko” which simulates an Ultrasparc III system. We have tested flexus-test-app using this system configuration. Note that the Solaris installation process may take 6 to 12 hours.

The installation process creates a big Simics disk image called **abisko-sol10-install.disk** and a state file called **abisko-sol10.state**. After verifying that disk has been generated correctly (last step from the manual on page 57) place the disk image file on the directory of your choice and copy **abisko-sol10.state** to the folder **FLEXUS_ROOT/flexus-test-app/checkpoints**. Modify the checkpoint_path variable in abisko-sol10.state to match the path where the disk image resides, and change the line **boot_command: "boot -v"** to **boot_command: "boot -rv"** in the same file.

Installing Linux on x86 machines

Chapter 6 of Simics x86-440BX Target guide provides instructions on how to install Linux on different x86 architectures. For our example we downloaded directly from Simics website a disk dump of a 'tango' simulated machine running Fedora Core 5 (http://www.simics.net/pub/simics/import/x86/tango1-fedora5.craff). More Linux disk images are available for download from Simics website.

Download the disk image into **FLEXUS_ROOT/flexus-test-app/checkpoints**. Verify that you are able to boot the machine with this disk image by running simics with tango-common.simics script, which can be found in SIMICS_ROOT/targets/x86-440bx directory.

Preparing the Test Application

After generating or acquiring a disk image of the system you want to simulate, you can boot the machine and launch flexus-test-app. The test application is distributed in source form and as a Simics disk image. The source code and input files are included in FLEXUS_HOME/flexus-test-app/src and FLEXUS_HOME/flexus-test-app/inputs, respectively. The test application binary and inputs are preloaded on Simics disk images located in **FLEXUS_HOME/flexus-test-app/checkpoints/**. "**flexus-test-app-disk_v9.craff**" has a Solaris10 compiled version of the application and "**flexus-test-app-disk_x86.craff**" an x86 compiled version.

Flexus includes a script which automates the process of booting and launching the test application. Go to **FLEXUS_HOME/flexus-test-app**. Edit the **prepare-test-app** script to select the architecture you are about to simulate, by setting the corresponding variable **IS_X86_ARCH** or **IS_V9_ARCH** to '1'. To prepare the test application run:

```
./prepare-test-app      (see also Appendix:A2, A3)
```

This script launches Simics to boot a single-CPU system and start the test application. If you wish to test a multiprocessor system, modify the @test_app_threads, the $num_cpus and the $megs_per_cpu lines in prepare-test-app-xxx.simics. Comments in the file illustrate the appropriate lines for an 4-CPU system. You should use CMPFlex in place of UniFlex and TraceCMPFlex in place of TraceFlex in the steps described in the remaining sections when simulating a multiprocessor.

At the end of the boot process, the scripts will automatically save a checkpoint of the system state in FLEXUS_HOME/flexus-test-app/checkpoints. The boot process may require 10-15 minutes.

Creating Flex-points

Once you have built the TraceFlex simulator and prepared flexus-test-app, you can create flex-points for timing simulation. Each flex-point contains a Simics checkpoint alongside Flexus cache hierarchy, branch predictor, and multiprocessor cache coherence state.

The flex-point creation process is carried out by two scripts. The first script runs flexus-test-app until it reaches the start of its second iteration (the second repetition of the image filtering process). This simulation excludes the initialization phase of flexus-test-app from timing measurements and warms caches and the branch predictor for flexus-test-app's first iteration. (The following command assumes Flexus has been installed with make install). To launch this process, run:

```
./create-initial-flexpoint TraceFlex     (see also Appendix:A3, A4, A5)
```

After 5-10 minutes, Simics will save an initial flex-point in **FLEXUS_HOME/flexus-testapp/checkpoints/flexpoints/flexpoint_000** and exit.

Once the initial flex-point is ready, a second script creates a sample of flex-points spaced every four million instructions throughout the second iteration of flexus-test-app. On a SPARC and x86 uniprocessor systems, the script should create additional 11 (12 for x86) flex-points. The flex-points are created by running:

```
./create-flexpoints TraceFlex     (see also Appendix:A3, A4, A5)
```

After another 10-15 minutes, all flex-points should be ready. The flex-points are each written to separate subdirectories under **FLEXUS_HOME/flexus-test-app/checkpoints/flexpoints**.

Running a Sampled Timing Simulation

With flex-points created, you are now ready to launch a timing simulation. The run-job script in **FLEXUS_HOME/flexus-test-app** automates the process of setting up and launching an individual Flexus simulation. The run-timing-jobs wrapper script launches run-job repeatedly for a range of flex-points. You will use this script to launch simulations for all the flex-points in our sample, as follows (see also Appendix:A5, A6, A7):

```
./run-timing-jobs 0 11 -ma -cfg OoO UniFlex.OoO (SPARC simulated machine)
./run-timing-jobs 0 12     -cfg InOrder UniFlex (x86 simulated machine)
```

The above command runs separate UniFlex simulations for flex-points 0 through 11. The –ma flag is required to launch Simics in micro-architecture mode, which is supported by Flexus only for SPARC simulated machines. The –cfg option indicates that configuration settings for Flexus should be loaded from the **FLEXUS_HOME/flexus-test-app/config/** directory. You can define your own configuration by making a new directory with all the required files in it, or you can modify the existing ones.

The simulations write their output files into a subdirectory of FLEXUS_HOME/flexus-test-app/runs. The subdirectory name is based on the current date and time, to allow multiple results to co-exist in the runs directory.

Aggregating and Examining Sampled Results

The timing simulation of each flex-point produces a separate Flexus statistics database. Each statistics database contains a wide variety of statistics (instruction counts, cache event counts, etc.) collected over one or more measurement regions, as controlled by the Flexus configuration. The scripts for the example experiment configure UniFlex to collect statistics for regions of 50,000 cycles each. For each flex-point, we will use the first two regions (100,000 cycles) for microarchitecture warmup and report the results collected in the third measurement region.

To report overall results, the individual databases must be aggregated into a single database. The aggregation is accomplished by the **stat-collapse** and **stat-sample** tools. **stat-collapse** selects and combines measurement regions within a single statistics database. **stat-sample** aggregates the output of stat-collapse across statistic databases and calculates standard deviations for confidence calculations. Both steps are automated by the sample-results script. To run this script:

```
cd FLEXUS_HOME/flexus-test-app/runs/<run name>
../../sample-results
```

These scripts will create a file called **stats_db.out.gz** in the current directory, which contains the aggregated results from all flex-points. **stat-sample** aggregates results in four ways: sum, average, count, and standard deviation. The results of each of these aggregation methods are included in the output statistics database. (Note: the following examples assume stat-manager is available on your path. If not, you must give the full path **FLEXUS_HOME/stat-manager/stat-manager**). Issue the command:

```
stat-manager list-measurements
```

To view the value of every statistic, summed over the flex-points, issue the command:

```
stat-manager print sum
```

stat-manager is also capable of performing calculations on statistics. For example, to print average IPC for the OoO uniprocessor:

```
stat-manager format-string "<EXPR:{sys-uarch-Commits}/{sys-cycles}>" avg
```

Similarly, for the InOrder uniprocessor:

```
stat-manager format-string "<EXPR:{sys-execute-Commits}/{sys-cycles}>" avg
```

Finally, stat-manager can process prepared report files. In these reports, stat-manager replaces place-holders for statistics and formulas with their values. To print a report of IPC and its associated 95% confidence interval:

```
stat-manager format ../../reports/ipc_OoO_uniproc.rpt     ".*"     - or -
stat-manager format ../../reports/ipc_InOrder_uniproc.rpt ".*"
```

To generate a report that includes a breakdown of execution time:

```
stat-manager format ../../reports/time_breakdown.rpt      "sum"
```

The output of the execution time breakdown report can be imported into the Excel spreadsheet in **FLEXUS_ROOT/flexus-test-app/reports/time_breakdown.xls** to graph the time breakdown.

To import the data:

1. Open time_breakdown.xls
2. Navigate to the "Data" tab
3. Run the preceding stat-manager command to generate the time breakdown report
4. Copy the output of the report to the Windows clipboard
5. Select cell A1 of the "Data" tab and paste the text of the report
6. (Excel 2002): Click the paste context menu and select "Use Text Import Wizard…". Click finish.
7. (Excel 2003): Select the pasted data, choose "Text to Columns…" from the "Data" menu. Click finish.

The report data should fit the shaded portion of the sheet.

Return to the "Time Breakdown" tab to view the graph. Further stat-manager reports and associated spreadsheets can be based on these examples.

Statistics for x86 simulation

In x86 simulated machines some statistics maintain two versions of counters. The first version has the same name as in SPARC simulated machines, while the second one has an "_x86" suffix. The purpose of this is to count correctly events that appear to be slightly different for x86 architectures, but also maintain the initial version of the counter when this makes sense. For example, "Nodes-execute-Commits_x86" and "Nodes-execute-Commits" count the committed instructions across all processors. An instruction that performs two memory accesses will increase the "Nodes-execute-Commits" counter twice but only once the "Nodes-execute-Commits_x86". Also, table walks made by the hardware will increase the "Nodes-execute-Commits" counter but not the "Nodes-execute-Commits_x86".

Appendix

The following section lists **some** of the warnings and errors that you may encounter during the process of compiling, setting up and running simulations with Flexus and Simics. Most of the warnings described in this section can be easily fixed or safely ignored. Having those as example you can then determine if or how much a different type of warning may affect your simulation results.

A.1 Compiler Warnings

Compiling different components and simulators of Flexus under GCC 4.1.2 you may encounter warnings of the following form, which you may safely ignore.

```
MemoryLoopbackImpl.v9_iface_gcc_o
flexus-3.0.0/components/Common/Transports/MemoryTransport.hpp:65:warning:
`Flexus::SharedTypes::<unnamed>::ExecuteStateTag' defined but not used
flexus-3.0.0/components/Common/Transports/MemoryTransport.hpp: 73:warning:
`Flexus::SharedTypes::<unnamed>::uArchStateTag' defined but not used
flexus-3.0.0/components/Common/Transports/MemoryTransport.hpp:
81:warning: `Flexus::SharedTypes::<unnamed>::MuxTag' defined but not used
flexus-3.0.0/components/Common/Transports/MemoryTransport.hpp:
89:warning: `Flexus::SharedTypes::<unnamed>::BusTag' defined but not used
flexus-3.0.0/components/Common/Transports/MemoryTransport.hpp:
98:warning: `Flexus::SharedTypes::<unnamed>::DirectoryEntryTag' defined
but not used
```

A.2 Warning during boot of the simulated machine

While booting Solaris 10 you may encounter the following warnings, which you can safely ignore.

```
Jun 2 17:13:50 abisko fmd[330]: libpkcs11:
/usr/lib/security/pkcs11_softtoken.so unexpected failure in ELF
signature verification. System may have been tampered with. Cannot
continue parsing /etc/crypto/pkcs11.conf
Jun  2 17:13:50 abisko fmd[330]: libpkcs11:
/usr/lib/security/pkcs11_softtoken.so unexpected failure in ELF
signature verification. System may have been tampered with. Cannot
continue parsing /etc/crypto/pkcs11.conf
```

Also, when booting Fedora Core 5 you may encounter the following warnings, which you can safely ignore.

```
Checking for hardware changes/etc/rc3.d/S05kudzu: line 23:  1143
Segmentation fault    /sbin/kudzu $KUDZU_ARGS             [FAILED]
```

A.3 Failing to save Simics checkpoints and Flexpoints

The scripts `prepare-test-app`, `create-initial-flexpoint` and `create-flexpoints` are saving simics checkpoints and flexpoints under the checkpoints/ directory. However, if a checkpoint is present in that directory from a previous run Simics will refuse to save the new checkpoint by overwriting the old one (the name of the checkpoints are going to match because they are generated by the same scripts). The same happens when saving flexpoints. Flexus will succeed in saving all the related flexus-state but Simics will fail saving the corresponding checkpoint. In order to generate new checkpoints/flexpoints either delete the old ones or transfer them to a different directory.

A.4 Scripts failing to load Flexus in Simics

The scripts `create-initial-flexpoint` and `create-flexpoints` will try to load in Simics the Flexus module, after searching for it in FLEXUS_ROOT/simulators/SIMULATOR_NAME/libflexus*. If there are two modules

there (e.g., libflexus_UniFlex.OoO_v9_iface_gcc.so and libflexus_UniFlex.OoO_v9_vverb_gcc.so) the scripts will exit complaining about not being able to pick one of the modules. You can either delete the modules that you don't want to use, or rename them.

A.5 Warnings and errors about missing parameters in Flexus

Whiles Flexus module gets loaded in Simics it initializes itself based on the value of the parameters provided. There are three ways to pass a parameter in Flexus. In order of decreasing priority these are: the configuration file provided in every run, the wiring file in the simulator directory, and finally any default values hard-wired in the code. Some parameters, due to their importance, are required to be set by the user, even if their default value is sufficient. If they are not set, either a warning message will appear on the screen – as the following one – and the simulation will use number in the wiring file (or the one hard-wired in the code), either the simulation will fail with an error message.

```
e.g., <ConfigurationManager.cpp:100> {0}- WARNING: -feeder:CMPwidth
(CMPWidth)  was not set in initializeParameters(), from the command
line, or from Simics).
```

Finally, another case where you may see warnings about parameters is when you try to define the value of a non-existing parameter, either because the name has been misspelled or the configuration file used matches a different simulator that uses the specified parameter.

A.6 Warnings from the OoO core

Some extremely infrequent behaviors/instructions of the Out-of-Order Ultrasparc core have not been taken under consideration and are either simulated partially or not at all. This doesn't affect the correctness of the simulation, since after committing an instruction Flexus' and Simics' state is compared, and if it is found to differ then Flexus synchronizes its state with Simics. During this process a lot of Flexus' state gets printed on the screen.

A.7 Warnings while trying to stop simulation

Flexus automatically tries to stop simulation and save results when it has reached the target number of simulated cycles. However, at that specific point Simics may not be able to be halted, requiring some more cycles (less than 1K cycles) before it can safely stop. This causes Flexus to issue multiple times the exit message of successfully saving the results and stopping Simics in consecutive cycles. The file keeping the results will not be affected by this behavior.