Lab 1 — Cache Simulation with Simics

Andreas Sandberg `<andreas.sandberg@it.uu.se>`

1 Introduction

The purpose of this assignment is to give insights into:

- how a cache works
- how different cache designs affect program execution
- how a program can be tuned for a specific cache configuration

You will extend the full system simulator Simics with a simple cache model and perform experiments with different programs and cache configurations.

You can (should) team up and work in groups of two. This lab assignment is examined in the computer lab. During the examination, you will be asked to demonstrate and explain your solutions.

1.1 Simics

Simics is a full system simulator platform that provides a controlled, deterministic and fully virtualized environment. Simics allows you to build your own virtual computer, or extend pre-configured models, based on microprocessors such as Alpha, ARM, Itanium, MIPS, Pentium, PowerPC, SPARC and x86. It is provided by Virtutech[1].

The simulated computer is referred to as the target machine and the computer running the simulator is referred to as the host machine. When referring to input given on the host, the target and to Simics the following prompts will be used **`host$`**, **`target#`** and **`simics>`** respectively. Output from Simics and the host machine will be printed with `this font`.

2 Getting started

2.1 Uppmax

In this lab we will be using the Os cluster[2] at Uppmax. Users of this cluster are allowed to run interactive jobs. To login on the cluster, connect with *SSH* to *os.uppmax.uu.se*, e.g. using:

```
host$ ssh -Y user-name@os.uppmax.uu.se.
```

You are now connected to one of the login nodes of the cluster. These nodes are only used to test code and submit jobs to the cluster.

[1] `http://www.virtutech.com/`
[2] `http://www.uppmax.uu.se/systems/os`

Note: If you start Simics on one of the login nodes, it will be terminated after about an hour.

To start a proper interactive job, run the following command on the login node:

```
host$ qsh -P g2010003 -l mem=2G -l h_rt=04:00:00
```

This will start an *xterm* that can run for a maximum of 4 hours and use a maximum of 2 GB of RAM. You may want to change these number if you expect to be working for more than 4 hours or need more memory. You should always include the **`-P g2010003`** when submitting related to this course, this ensures that the correct project is billed for the used CPU time.

2.2 Simics

Simics uses workspaces to manage user projects. A workspace can be thought of as a local Simics installation, it contains symbolic links to all the relevant Simics binaries and documentation. It can also contain custom modules that implement hardware components or other parts of the simulation.

Login to Os and execute `~ansan501/avdark/2010/init_lab1.sh` to create your own workspace. This script creates a new workspace in `~/avdark/lab1` and adds the modules necessary to simulate the x86-architecture. The script also installs a small cache simulator in `~/avdark/lab1/modules/avdark-cache`, but more on that later.

To start Simics, schedule an xterm on the Os cluster using the instructions in subsection 2.1. In the xterm that was started by the previous step, execute the following:

```
host$ cd ~/avdark/lab1
host$ ./simics ./targets/x86-440bx/cosmo-common.simics
simics> run
```

This starts a simulation of a 64-bit x86 machine running Fedora Core 5. When the target machine has booted, login with the user-name `root` and the password `simics`.

The boot process takes a few minutes, so be a good student and read the next section while the machine boots. Pay particular interest to the *snapshot* functionality, this will save a lot of time!

3 Simics

3.1 Basics

Simics can be given commands at the Simics prompt: **`simics>`**. Simics has several commands that can be used to control the simulator, view current machine state, set breakpoints and more. A short description of some simple commands needed for this assignment follows below. Simics command line interface supports tab-completion. If you press the tab key, Simics will automatically expand the command you are writing. If not, it is ambiguous and pressing the tab key again will give you a list of alternatives.

help ***[CATEGORY]*** Displays help on various topics. Displays available topics if executed without parameters. The *CATEGORY* argument can be either a topic from the topic list or a command.

quit Exits Simics.

run *[COUNT]* Starts, or continues, the execution. The optional argument *COUNT* is the number of instructions to execute, if *COUNT* is omitted, the simulation continues until it is explicitly stopped.

stop Stop the simulation. The simulation may be resumed using the *run* command.

write-configuration *NAME* Write a snapshot of a machines state to disk. The snapshot allows you to restart execution at a given point. To start Simics from a snapshot, use `host$ ./simics -c NAME`, where *NAME* is the name of the snapshot.

load-module *NAME* Load the module *NAME*.

unload-module *NAME* Unload the module *NAME*.

list-modules List modules that are loaded or can be loaded.

enable-magic-breakpoint Enables magic breakpoints. This will cause Simics to stop the execution when a magic instruction is executed in the target machine.

disable-magic-breakpoint Disables magic breakpoints.

3.2 Accessing the host file system

There is a module called `hostfs` in Simics that allows the target machine to access the file system of the host machine. This module is loaded by default in the workspace used in the lab.

To access the host file system from the target you have to mount the file system. Execute the following command on the target machine:

```
target# mount /host
```

This mounts the host file system on `/host`. For example, execute `target# ls /host/sw` to list `/sw` on the host machine.

Note: The kernel caches file system data, so you might not get the latest version of a file when you read it on the target after writing to it from the host. If this is a problem, create a new file with a new name instead of updating the old file.

3.3 The Cache Model

The cache model is a custom extension to Simics. It simulates a direct mapped level one data cache. The source of the cache model is located in `~/avdark/lab1/modules/avdark-cache`. To load the module and connect it to Simics' memory system, run `simics> setup-avdark-cache`. The cache model has its own set of commands, use `simics> help avdark-cache` after loading the module to list them.

```
simics> setup-avdark-cache
simics> help avdark-cache
[...]

Command List
  [...]
  Commands
    dbg-disable          disable dbg printing
    dbg-enable           enable dbg printing
    disable              disable the cache
    do-access            do a read/write access.
    enable               enable the cache
    flush                flush the cache
    info                 print the cache information
    print-internals      print the data array
    reset-statistics     reset statistics
    resize               resize the cache
    statistics           print statistics
[...]
```

These commands are called object commands and can only be executed on an object. When the avdark-cache model is instantiated, it will be called *dc0*. The avdark-commands are executed on the *dc0* object. Try the **`simics> dc0.info`** command to get the info related to the *dc0* object. Note that Simics supports tab completion, type **`simics> dc0.`** and press tab twice to get a list of commands applicable to the dc0 object.

Object commands are defined by the Python script `commands.py` in the source directory. Another Python script, `gcommands.py`, is used to define global commands, such as **`setup-avdark-cache`**.

4 Measuring Miss Ratio for the RADIX application

In this assignment, your will use the RADIX application as test program for cache miss ratios. The RADIX application is located in the directory: `~/avdark/lab1/radix`

The procedure to measure the miss ratio is as follows: Start Simics as described above and mount the host file system if you have not done so already. Then start the RADIX program on the target machine using the host file system. I suggest that you copy the RADIX program to the targets file system before you run it.

```
simics> enable-magic-breakpoint
simics> continue
target# cp /host/home/user-name/avdark/lab1/radix/radix ./
target# ./radix -n 100000
```

The RADIX program contains so called magic breakpoints where Simics stops, the command **`enable-magic-breakpoint`** makes Simics catch such breakpoints. The idea here is that a magic breakpoint is located before the computation in RADIX and another after the computation. This makes it possible to execute RADIX and stop at the first breakpoint. Enable the cache model, continue the execution and, finally, when the second breakpoint is captured, print the cache statistics.

5 Modifying the cache model

Edit the file `avdark-cache.c` in the source directory (`~/avdark/lab1/modules/avdark-cache`) to modify the cache model. To rebuild the module change directory to `~/avdark/lab1` and run **`host$ make`**. If you prefer not to change the working directory use **`host$ make -C ~/avdark/lab1`**. This can of course also be done from Emacs. Press M-x and type *compile* followed by enter, edit the command line if needed and then press enter again.

The internals of the cache model are fairly simple, most of the code is just Simics glue code. Your assignment boils down to rearranging the data line array or/and fix the tag and index computations.

The `dc_dbg_log` function can be used to print help information (but only when debug has been enabled, see the **`dbg-enable`** command). The command **`do-access`** can be used to test your cache. It performs an access in the cache (**`simics> help dc0.do-access`** for more information). In combination with the **`dbg-enable`** and **`print-internals`** commands, this is a powerful method of debugging your cache simulator.

The `avdark_cache_operate` function will be called on all read and write requests (when the cache is enabled, see the enable command). It is also called when the **`do-access`** command is executed (and the cache is enabled).

All important functions are explained in the source code. It is recommended that you spend some time with the original cache model to get a basic understanding of it before you start to modify it.

6 The Assignments

6.1 Simulating an associative cache

At the moment the cache model is only direct mapped. Modify the cache model so that it can be configured as both a direct mapped (i.e. 1-way) and a 2-way associative data cache. The 2-way associative cache should use the LRU-replacement policy. Note that the cache model never handles actual data, Simics takes care of that. The cache model only contains tags and valid bits.

6.1.1 Evaluation

- Describe the modifications made to the cache model (by detailing the source code).
- Run 2 or 3 examples with different cache parameters to show those modifications (i.e. prove that it works). Use the **`do-access`** command on the cache object (*dc0*) to "insert" accesses into the cache.

6.2 Miss ratio measurements

Test the miss ratios for the RADIX program for the following cache settings:

Use RADIX with the -n 100000 option, i.e. run **`target# ./radix -n 100000`** on the target machine. This makes RADIX sort 100000 keys. The RADIX binary is located in `~/avdark/lab1/radix`.

6.2.1 Evaluation

1. Live-run some examples with different cache parameters.
2. Examine and draw some conclusions from the cache-behavior of RADIX.

Size	Block Size	Associativity
16 kB	32	1
16 kB	32	2
32 kB	16	1
32 kB	32	1
32 kB	64	1
32 kB	16	2
32 kB	32	2
32 kB	64	2
64 kB	32	1
64 kB	32	2

Table 1: Cache configurations

Note: You don't need to test all configurations in Table 1, 6 different configurations should be enough to be able to draw some conclusion. Explain how selected your cache configurations in that case.

6.3 Improving Cache Performance (optional, bonus)

In this assignment you should try to improve the cache performance of a given program example. The program is a simple matrix multiplication. Try to rewrite the program so that the cache miss ratio is decreased. Use some of the methods discussed in the course book to reduce the miss ratio of the program.

You may not change the size of the matrix or use less precision, but otherwise you are free to experiment with loop-indices, matrix-transponation, blocking etc. The multiplication should be correct, though. Also, you must use *gcc* when compiling it.

Use Simics and your modified cache model to measure the miss ratio before and after your modifications. Note that the Simics model does not stall the processor, you will therefore not see any improvement in runtime on the simulated system. During the measurements use a *2-way* associative, *64 kB* cache with a block size of *64 B*. See subsection 3.2 for information about transferring files to the target, pay special attention to the note about file system caches.

When experimenting with the optimizations, it might be a good idea to run directly on the host. In that case, change the matrix parameter SIZE so that the unoptimized version runs for about 20 seconds. Then try to minimize the execution time (on the host) without changing the SIZE parameter.

The original program is located in `~/avdark/lab1/multiply`. Copy multiply.c and Makefile to a working directory and . . . have fun!

The original code has built in support for verifying the results. You may activate the verification code using the **`-v`** option to the binary. It is a good idea to run the verifications on the host machine instead of the simulated machine to save some time.

6.3.1 Evaluation

1. Show the optimizations done to the multiplication program and explain them.
2. Verify all your numbers by running simulations and the different versions of the multiplication program.

3. Show that your implementation is an improvement.