CS152 Laboratory Exercise 2

Professor: Krste Asanovic
TA: Christopher Celio
Department of Electrical Engineering & Computer Science
University of California, Berkeley

February 16, 2011

1 Introduction and goals

The goal of this laboratory assignment is to allow you to conduct some simple memory hierarchy experiments in the Simics simulation environment. Using the g-cache cache simulator module, you will collect cache statistics and make some architectural recommendations based on the results.

The lab has two sections, a directed portion and an open–ended portion. Everyone will do the directed portion the same way, and grades will be assigned based on correctness. The open–ended portion will allow you to pursue more creative investigations, and your grade will be based on the effort made to complete the task or the arguments you provide in support of your ideas.

Students are encouraged to discuss solutions to the lab assignments with other students, but must run through the directed portion of the lab by themselves and turn in their own lab report. For the open-ended portion of each lab, students can work individually or in groups of two or three. Any open-ended lab assignment completed as a group should be written up and handed in separately. Students are free to take part in different groups for different lab assignments.

You are only required to do one of the open ended assignments. These assignments are generally starting points or suggestions. Alternatively, you can propose and complete your own open ended project as long as it is sufficiently rigorous. If you feel uncertain about the rigor of a proposal, feel free to consult the TA or professor.

1.1 Tools and conventions

This lab assumes you have completed the first laboratory assignment. While we will re-include all the files used for both labs in this lab's distribution bundle for your convenience, there will be points where you can save time by reusing checkpoints and such created in the first lab.

Furthermore, we will assume that you remember all the commands used in the first lab for controlling Simics simulation. If you feel any confusion about these points, feel free to consult the first lab guide or the Simics User Guide.

Several new tools will be introduced in this lab, mainly having to do with the g-cache module. We will also introduce a new mode of Simics operation (`-stall`). For a more thorough explanation of these items, you can refer to the Simics User Guide chapter 18.

1.2 Graded Items

You will turn a hard copy of your results to the professor or TA. Please label each section of the results clearly. The following items need to be turned in for evaluation:

1. Problem 2.2: simple cache statistics for each benchmark and answers
2. Problem 2.3: suggested working sets and evidence
3. Problem 2.4: statistics and answers
4. Problem 2.5: complex cache statistics and answers
5. Problem 3.1/3.2/3.3/3.4 modifications and evaluations (include source code if required)

2 Directed Portion

2.1 General methodology

At its core, the Simics software is an instruction set architecture simulator, not a machine performance simulator. While Simics presents an interface to the OS and programs running within it that makes the target machine appear to be a normal machine, in actuality many of the simulated machine's functions are idealized far beyond the capabilities of a real machine. For example, in normal execution mode, main memory accesses in Simics appear to take zero cycles.

However, it is possible to attach a cache simulator module to Simics and use it to gauge a program's cache performance or the effectiveness of a particular cache hierarchy. The cache simulation module included with Simics is called `g-cache`. These modules can be linked into hierarchies, connected to multiple processors and have a variety of adjustable parameters. `g-cache`s are configured and attached to the memory hierarchy by means of simple Simics scripts containing python commands (the @ prefix tells Simics to interpret a statement as python).

One of these `g-cache` parameters is the delay accrued by a memory request as it passes down through each level of the hierarchy to the simulated memory interface. By running Simics in `-stall` mode, users can force the delayed execution of instructions according to whether or not the data the instructions require is contained in caches or memory. Simultaneously, the cache modules record statistics about the memory requests which have accessed them. `-stall` mode is a more detailed mode of simulation, and therefore noticeably slows down the operating speed of the target machine.

A further methodological detail is that when the caches are first attached to the simulation, they are empty. Any statistics recorded from them initially will only reflect the compulsory misses encountered as they fill up. For this reason, it is necessary to 'warm' the caches by running the intended benchmark for millions of instructions before the true cache performance statistics can be collected.

Thus, the general methodology of this lab takes the following form:

1. Run the simulation at full speed to get the file system loaded, the benchmarks set up, etc.
2. Checkpoint the simulation
3. Restart simulation in `-stall` mode

4. Load the caches using a .simics script
5. Warm the caches by executing benchmark code
6. Pause or breakpoint the simulation
7. Reset all the caches' statistics
8. Continue to run the benchmark and collect real cache data

Data from the cache modules can take several forms. `<cache name>.info` reports the configuration of the cache, `<cache name>.status` displays the current value of every cache line, `<cache name>.statistics` reports cache performance statistics, and `<cache name>.add-profiler` adds a profiler module which tracks cache misses on a per instruction or address basis.

You can use any of the `t7400-{1,2,...,12}.eecs` machines to complete this lab assignment. Get started early, because the simulations in this exercise will take a significant amount of time to run, and this will only be exacerbated by contention for the servers.

2.2 Collecting statistics from a simple cache

You should untar the file `benchmarks-1.tar` distributed with this Lab (`tar -xvf benchmarks-1.tar`) and make sure the three binary and three input files are available in your `simics-workspace` directory. The binary files for this lab are the same as the ones used in Lab 1 (`bzip2_sparc`, `mcf_sparc`, `soplex_sparc`).

Start Simics, using either the `targets/sunfire/bagle-common.simics` script or a checkpoint you created in the previous lab. Make sure the host filesystem or workspace is mounted and that the 3 benchmark binaries and 3 benchmark input files are copied into the target machine. To save time in the following sections, you should create a checkpoint that has all the files loaded, or possibly a checkpoint at each of the initial magic breakpoints in the benchmark programs.

For each benchmark:

- Start Simics in stall mode and load the proper configuration file
  ```
  host$ ./simics -stall -c <configuration name>
  ```
- Execute the following commands in order to ensure the proper operation of the benchmarks and caches:
  ```
  simics> magic-break-enable
  simics> istc-disable
  simics> dstc-disable
  ```
- Load the cache module into the simulation (a .simics file is simply a sequence of Simics commands).
  ```
  simics> run-command-file add-1cache-bagle-2.2.simics
  ```
- Look at the configuration of the cache you have loaded, and compare this to the contents of the .simics file you just executed. Observe how the cache is configured with simple declarative statements in Python.
  ```
  simics> cache.info
  ```

- Run one of the benchmark programs.

  ```
  target# ./bzip2_sparc -z input.jpg
  ```
  or
  ```
  target# ./mcf_sparc input.in
  ```
  or
  ```
  target# ./soplex_sparc input.mps
  ```

- It will quickly reach a magic breakpoint and the simulation will pause.

- Run for 100,000,000 instructions in order to warm the cache. This may take a few minutes as Simics is now running in a more detailed simulation mode.
  ```
  simics>  c 100_000_000
  ```

- Reset the cache statistics to clear the data recorded for the warming period. Then record data for the next million instructions.
  ```
  simics> cache.reset-statistics
  simics> c 1_000_000
  ```

- Display the recorded statistics
  ```
  simics> cache.statistics
  ```

- `Continue` the simulation, halt the benchmark, and run another, remembering to clear the cache statistics before you collect more data.

For each benchmark, record the hit ratio for all types of memory requests (instruction fetch, data read, data write). Which benchmark has the best cache performance? Which has the worst?

2.3 Determining benchmark working set size

Your task in this section is to determine the working set size of each of the benchmarks by varying the size of the simple unified cache (`add-1cache-bagle-2.2.simics`) used in the previous section. Record the measurements you make that support your claim. Which benchmark seems to have the largest working set, and how big is it?

Note that once a cache has been configured to be a certain size, changing its size parameters will have no effect on simulated performance. You must either restart Simics and reattach the cache after modification, or create a new cache and attach it instead. The line

```
simics> @conf.phys_mem.timing_model = conf.<cache name>
```

can be used to switch which cache model is connected to the memory hierarchy.

2.4 Minimal instruction and data caches

For this section you will use the spilt instruction and data caches defined in `add-2cache-bagle-2.4.simics`. Examine the .simics file and note that each cache contains only one line! Also note that there is a new module present: an id-splitter that routes instructions to the appropriate L1 cache. Run the benchmarks and record their performance on this minimal cache. What can you learn about the relative locality of data vs instruction accesses?

Modify the cache so that it still only has one line, but make this line 4 times longer (128 bytes). Record how this changes performance. What does performance under the improved cache indicate

about the spatial locality of each benchmark? What can you learn about the relative locality of data vs instruction accesses?

2.5 Collecting statistics from a cache hierarchy

In this section we will attach a more complicated cache hierarchy to our simulation in order to conduct a more realistic study of the cache behavior of these benchmarks. This new hierarchy is defined in the file `add-2cache-bagle-2.5.simics`. The hierarchy has a split L1 data cache (`dc`) and L1 instruction cache (`ic`), both of which are connected to a larger L2 cache (`l2c`). Note that there are also a new modules present: a transaction-staller that simulates the delay incurred by accesses to main memory. This configuration is similar to the one illustrated on Simics User Guide p. 200.

Note that memory accesses are now diverted from physical memory to the id-splitter, since it is now at the top of the hierarchy. Make sure that as you connect reconfigured caches into the hierarchy you preserve the flow of timing delay down from the id-splitter all the way to the transaction-staller.

Yours tasks for this section are to:

- Collect L1 and L2 hit ratio statistics for the three benchmarks running on the complex cache hierarchy. How effective are the L2 caches at collecting misses from the L1 level?
- Modify the L2 cache (`l2c`) so that it is only twice the size of each L1 cache (i.e. 8Kb). Pick one benchmark, and report how its L2 statistics change as a result of this size modification. Do the same for an L2 cache that is twice the size of the original (i.e. 1Mb). What can you conclude from these results?
- Look at the stall penalties assigned to each cache level (3 cycles in L1, 10 cycles in L2, 200 cycle to memory). Calculate the average memory access time of each of the three hierarchies that you have collected data for.
- Assume that the clock cycle time for the Bagle machine is 0.25 ns, that there are 1.3 memory references per instruction, and that with a perfect cache the machine would have a CPI of 1.5. Further, assume that since a cache hit is on the critical path, changing the cache size means that the clock cycle time must be scaled to accommodate the new size. For the 1MB cache, clock cycle time is scaled up to 1.2x the original, and for the 8Kb cache the cycle time can be scaled down to 0.5x the original[1]. Calculate the relative performance in terms of CPU time for the larger and smaller caches compared to the original. Which design do you recommend? (See H&P 4th edition page C-18 if you get stuck.)

3 Open-ended Portion

3.1 Tuning code to fit a memory hierarchy

Matrix multiplication is a task common to a wide variety of scientific and machine learning programs. Often, the performance of the computationally intensive core of these codes is based almost

[1] In reality, the L2 cache size will affect CPI more than cycle time because accesses are pipelined. But, for the purposes of this exercise, suspend disbelief and assume it affects cycle time.

solely on the efficient execution of this common operation. Matrices used in such calculations can be quite large and so cache performance often has a direct impact on overall program performance.

To address the problem of multiplying large matrices, many scientific codes make use of a *blocked matrix multiply algorithm*. The algorithm divides the matrix multiplication task into smaller size chunks which only use a subset of the data contained in large matrix. By adjusting the block size, users can control the amount of data being operated on at any given time. More information about the specific mechanics of this algorithm can be found on the web.

You have been given some GEneralized blocked Matrix Multiply code (`gemm-3.1.c`) and a cache hierarchy (`add-2cache-bagle-3.1.simics`). You assignment is to find a block size that makes optimal use of the given hierarchy. You are not allowed to vary any of the cache parameters, or any statement which is `#defined` in the C source file (though you may examine them). In other words, limit your modifications to main() and the block size parameters, which this code helpfully takes as command line arguments.

Remember that you will have to compile the gemm code inside the target machine for it to run, and that this should be done with Simics running in fast mode:

```
target# gcc -I/host/share/instsww/pkg/virtutech/simics-3.0.30/src/include
gemm-3.1.c -o mix
```

Also, remember that `gcc` within the Bagle machine only accepts ANSI C.

Report on your selected blocking parameters and any statistics you gathered or calculations you made that prove they are the best settings for your cache hierarchy.

3.2 Designing a memory hierarchy for an important benchmark

This study is essentially the opposite of the one presented in section 3.1. Instead of modifying the blocked matrix multiple code to better fit in the cache, you will modify the cache to better suit a specific implementation of the blocked matrix multiply code. This is analogous to creating a custom cache hierarchy for a special–purpose machine. Use the source code `gemm-3.2.c` (you may not modify it, but you may examine it) and the cache hierarchy file `add-2cache-bagle-3.2.simics` (you may modify it as much as you like, even to the point of adding new caches). Remember you will have to compile the gemm code within the target machine as described in 3.1.

You may also want to use the delay, power, and area statistics reported by CACTI (http://quid.hpl.hp.com:9081/cacti/) as further motivation for your chosen configuration. Use 1 bank and technology node of 65nm. The statistics you are most interested in will be in the top central column. Creating an enormous cache is useless if the access time, power, and area overheads are prohibitive.

Report on the configuration you select and any performance statistics you recorded or calculations you made that prove it is well suited to this BGEMM benchmark.

3.3 Use your own code

Perform a study after the fashion of either 3.1 or 3.2, but use your own code, perhaps from another project you have worked on. You must submit the relevant portion of source code (original and modified) and an explanation of its operation along with your report. Remember that you must

be able to compile this code inside the simulated target machine to perform this study. C++ code is also acceptable (compile with `g++`).

3.4 Study the effect of other cache parameters

Complete an analysis of the effect of other cache parameters on the performance of the previously–used benchmarks. This study should be thorough and you should report the effect of changing these parameters in multiple cache configurations. Consult the Simics User Guide Chapter 18 for more information on the available cache parameters. Some suggestions for further study:

- Complete an analysis of the differences in memory traffic between a hierarchy with write through versus write back caches. Remember to adjust your write allocation scheme.
- Experiment with different replacement policies your caches ('random', 'lru' and 'cyclic') . Does the optimal strategy seem to vary more between benchmarks or between cache configurations?
- Study the effect of virtual versus physical tagging/indexing. Is there a significant performance difference? Does it matter which level of the hierarchy a certain method is used at? How can the overhead associated with the translation be dealt with within the cache hierarchy?

For any of these studies, make sure to report all the statistics you gathered and calculations you made to reach your conclusions.