LAB 3 - Caches

Assessment: 7% of the total course mark.

---

1 General Instructions

- ◇ start by accepting the assignment link, which will create the repo for you on your github. Then clone this repo into the VM.
- ◇ Make sure you use the provided VM for developing and running the labs. We will not help debugging any other setup.
- ◇ In the lab room machines, you can login in to the VM with your group number. You will be asked to enter a password for the first time. Feel free to copy the VM into your own laptop to work on the labs at your time.
- ◇ clone the repository into a folder named macsim-4dm4-lab2; all instructions in this document assumes this folder as the main repo folder.
- ◇ All the files you are asked to produce or modify (including source, configuration, trace files, and report) should be placed under labs/lab3 folder.
- ◇ The submission of the code, output files, and the lab report should be done by pushing to the repository before the deadline time.
- ◇ In addition to the code modifications and output files, you are asked to prepare a pdf report of all the steps and answering the questions within the lab document. The first page of the report should be a declaration of contribution where each of the two members of the group lists clearly what they have done.
- ◇ In addition to the declaration of contribution, we will be using github commit history to track the contributions of the group. Therefore, it is very critical to commit and push updates one at a time and do not wait until you finish the full lab to push once. This will be considered a red flag.
- ◇ Finally, please make sure you only push the required files, the report, the parameter files you modify for each step, plus any source files you modified. Log files and benchmark source files are not required.

2 Submission Deadline

The deadline for lab2 is **Tuesday Oct 31th, 11:59PM**. Please note that this is a programmed deadline in the environment, so make sure you submit in time since you will not be able to submit after that dictated deadline.

2.1 Importing the starter code of the lab

You should follow exactly the same process you have been following in past labs to import the starter code from the following invitation link and to create the project:

https://classroom.github.com/a/RZI6oIZj

3 Octopus Simulator Introduction

In this lab, we are using our in-house cache simulator, Octopus. Octopus enables you to configure all the cache parameters we discussed in lectures and is designed to be highly modular and extensible to be able to add modules/designs into it, which fits into the project later on.

3.1 Running

To run Octopus you need to invoke the script run.sh. Opening run.sh script, you see it is composed of two commands:

```
./waf configure
```

```
./waf --run "scratch/MultiCoreSimulator
--CfgFile=./tools/x86_trace_generator/LAB3_Configurations/lab3_1_8KB_64B_1_RANDOM_32KB_64_2_LRU.xml
--BMsPath=$1 --LogFileGenEnable=0"
```

Octopus currently uses waf flow to build and run the simualtor. You can read more about this in the README file. You do not need to worry too much about how does waf works, but if you are curious, you can find more information in their website here: https://waf.io/apidocs/tutorial.html

Two important input paramters to understand in the run command:

- ◇ CfgFile: This is the configuration file that sets the parameters of the cache hierarchy. Octopus takes an XML configuration file. You can look into the example invoked into the above command (lab3_1_8KB_64B_1_RANDOM_32KB_64_2_LRU.xml) and understand the different passed parameters as you will sweep some of them in this lab. This file configures Octopus to run:
  - a single core with
  - a 8KB L1 direct-mapped cache that has a 64B cache line size and a random replacement policy and:
  - a $32KB$ L2 2-way set-associative cache with a 64B cache line size and an LRU replacement policy.
  - Finally a main memory of a 250 cycles fixed latency.
- ◇ BMsPath: is the directory path where the BMs (memory traces) are located. Octopus accepts memory traces as input to simulate. The run script takes this as input, this is what it has the $1 afterwards as an input argument. Open the file under tools/x86_trace_generator/LAB3_BMs/cache_test_4KB_4_1 to see the format of this trace.

It is simply representing the interface of memory requests going out of the pipeline to the memory hierarchy. everyline represents a single memory request, where the first item is the memory address in hex and the third item is the type (read or write). You can ignore the '1' in the second item and the '0' in the forth item for now. More about how these traces are generated is discussed in next subsection.

3.2 Memory Trace Generation

We modified the trace_generator.cpp pintool from MacSim to also output memory traces for Octopus. You will find this along with the pintool under tools/x86_trace_generator folder. The benchmarks you will be using/generating is also in tools/x86_trace_generator/LAB3_BMs; under that folder you will find a baseline BM named as: cache_test_4KB_4_1.c. **Spend sometime understanding the source code of this simple C program as you will be asked to configure for various parts of this lab.**

3.3 Naming convention of configuration and BM source files

Please pay attention to the naming convention of the configuration files as well as the BM files as you will be asked to configure either or them or both throughout the lab experiments. You have to save the ones you modify/configure and submit with your lab submission using the naming convention as follows:

For the benchmark source code, note the naming convention of the file (cache_test_4KB_4_1.c) as we are setting in this test the following parameters (using defines in the top of the file):

- ◇ #define MEMORYPOOL_SIZE 4*KB
- ◇ #define STRIDE 4
- ◇ #define NUM_ITERATIONS 1

For the configuration file, note also the naming convention that sets the name with the following parameters in order: number of cores, L1 size, L1 cache line size, L1 number of ways, L1 replacement policy, then L2 size, L2 cache line size, L2 nubmer of ways, and finally L2 replacement policy: lab3_1_8KB_64B_1_RANDOM_32KB_64_2_LRU.xml

You must follow these naming conventions.

Before running the pintool, you need to update your ~/.bashrc file to point to the new location of pin. similar to Lab1, you do this by modifying the line where you point to pin at the end of the file to the following:

```
export PIN_HOME=/home/[USER]/lab3-[YOUR GROUP NAME/NUMBER]/
tools/x86_trace_generator/pin-3.13-98189-g60a6ef199-gcc-linux
```

Then invoke your bashrc file to reflect the effects:

```
source  /.bashrc
```

To make your life easier, we have provided a script named compile_and_run.sh in the main directory of the lab that takes as input your C file that you want to create the trace from,

invokes pintool to create the memory trace, and then invokes the run script to run the simulator. an example of invoking the script for the given benchmark cache_test_4KB_4_1.c is as follows:

```
source compile_and_run.sh
tools/x86_trace_generator/LAB3_Configurations/lab3_1_8KB_64B_1_RANDOM_32KB_64_2_LRU.xml
tools/x86_trace_generator/LAB3_BMs/cache_test_4KB_4_1.c
```

3.4 Simulation Output

Octopus output a detailed output log for all memory requests of the experiment your run. As you can see from the compile_and_run.sh script, this output file (named labdata.csv) can be found on the same directory as the memory trace of this benchmark. If you look in to the output file, you will find an example snippet as in Table 1. In this output, columns have the following meaning in their order:

1. message/request ID
2. address in hex
3. event for this request as it traverses the cache hierarchy
4. the meaning/explanatio of the event
5. coreID or LLC
6. cycle value that this event happened (it will have cycle written in last column) or in case it is a boolean value, it is either 0 or 1 (for example to indicate whether it is a hit or miss)

In addition to this detailed log, the compile_and_run.sh script prints a status summary in the summary.txt file in the same directoy as the output file including total number of requests, total hits, and total misses.

1	5588254222a0	add2q_l	L1: The message is added to the processing queue from the core	0	2	cycle
1	5588254222a0	Miss?	L1: Was the request a miss?	0	1	boolean
1	5588254222a0	MSIredy	L1: Protocol determines message is now ready to be processed	0	2	cycle
1	5588254222a0	msgProc	L1: message is taken from the L1's processing queue to be processed	0	2	cycle
1	5588254222a0	add_req	L1: the core's controller processes the message -¿ miss -¿ sends req to LLC	0	2	cycle
1	5588254222a0	add_req	L1: the core's controller processes the message -¿ miss -¿ sends req to LLC	0	3	cycle
2	5588254222a4	add2q_l	L1: The message is added to the processing queue from the core	0	3	cycle
1	5588254222a0	add2q_u	L1 receives message from the bus	0	4	cycle
3	5588254222a8	add2q_l	L1: The message is added to the processing queue from the core	0	4	cycle
1	5588254222a0	add2q_l	LLC: The request message is added to the LLC's processing queue	LLC	4	cycle

Table 1: Example of labdata output logger from Octopus

4 Cache and Locality Basics

4.1 Paper-and-Pencil Analysis

Consider the following C code snippet, where mempool is an array of integers that has the total size of MEMORYPOOL_SIZE (i.e. 4KB in this snippet):

```
#define KB 1024
#define MEMORYPOOL_SIZE 4*KB
#define STRIDE 4
#define NUM_ITERATIONS 1

for (j =0; j<NUM_ITERATIONS;j++)
   for (i = 0; i < MEMORYPOOL_SIZE/4; i+=STRIDE/4) {
      sum+=mempool[i];

}
```

- ◇ (a) Assume you have a 32-bit address and an 8KB direct mapped cache with a 64B cacheline size Calculate how many bits (and which bits are then in the address) are dedicated for offset, index, and tag. Calculate your tag overhead assuming you also have to consider a valid bit for each cache line.
- ◇ (b) Assume the address of mempool[0] is 0x0; calculate the hit and miss rate of this code snippet (Assuming all memory accesses are only coming from accessing mempool). **Show your steps.**
- ◇ (c) What type of locality (spatial or temporal, both, or none) is exploited in this code?
- ◇ (d) Assume now you change the NUM_ITERATIONS to 10, what is the hit rate? show your steps. What type of locality (spatial or temporal, both, or none) is exploited now?

4.2 Simulations

You are given the benchmark that implements this pattern in Section 4.1 under:
tools/x86_trace_generator/LAB3_BMs/cache_test_4KB_4_1.c,

- ◇ Double check that the BM configuration through the defines matches the pattern in this previous section for the L1 cache. For the L2 cache, assume a 16KB L2 2-way set-associative cache with a 64B cache line size and an LRU replacement policy.
- ◇ double check that the configuration file under tools/x86_trace_generator/LAB3_Configurations and you pass to the script also has the same cache setup as in Section 4.1 (4KB, 64B line size, and direct-mapped). **You need to modify this file to make OutOfOrderStages="1"** (This has to be the set value to use In-Order simulation for all experiments).
- ◇ compile and run that application by invoking the compile_and_run.sh script as explained earlier.

- ◇ using the data in the summary.txt file, calculate you miss and hit rates.
- ◇ compare your results from the simulation with the paper-and-pencil analysis and comment on this comparison.
- ◇ please note that since in Section 4.1 you use two settings for the NUM_ITERATIONS. It means that you need two benchmarks for this experiment, one for NUM_ITERATIONS 0 and the other for NUM_ITERATIONS 10. And you need to compare the output of each experiment with its corresponding analysis from Section 4.1.

5 Spatial Locality

For the same code snippet in Section 4.1, with the following configurations:

```
#define KB 1024
#define MEMORYPOOL_SIZE 4*KB
#define NUM_ITERATIONS 1
```

but with sweeping $STRIDE$ with the following values: $1, 2, 4, 8, 16, 32$, and 64.

- ◇ For each $STRIDE$ value, use the same paper-and-pencil analysis to obtain the hit/miss rate. Put the results in a csv file named spatial_locality_analytical.csv with the format:
  `stride,num_requests,num_hits,num_misses,hit_rate,execution_time`

  Note execution time is not required in the analytical solutions; you can find it as an output of compile_and_run.sh script.
- ◇ For each $STRIDE$ value, name the C source file accordingly and run the simulation with the same configuration file as in the past section.
- ◇ For each simulation, put the results in a csv file named spatial_locality_experimental.csv with the same format as above.

6 Cache Line Size

For the same code snippet in Section 4.1, with the following configurations:

```
#define KB 1024
#define MEMORYPOOL_SIZE 4*KB
#define STRIDE 4
#define NUM_ITERATIONS 1
```

Assuming also same cache configuration as in Section 4.1 but we are going to sweep cache line size. Assume a cache line size of $8, 16, 64$, and 128

- ◇ For each line size value, use the same paper-and-pencil analysis to obtain the hit/miss rate. Put the results in a csv file named line_size_analytical.csv with the format:
  `line_size,num_requests,num_hits,num_misses,hit_rate,execution_time`

- ◇ For each size value, configure the xml file and name the configuration file accordingly and run the simulation with the same configuration file as in the past section.
- ◇ For each simulation, put the results in a csv file named line_size_experimental.csv with the same format as above.

7 Cache Capacity

For the same code snippet in Section 4.1, with the following configurations:

```
#define KB 1024
#define STRIDE 4
#define NUM_ITERATIONS 2
```

but with sweeping $MEMORYPOOL\_SIZE$ with the following values: $4KB, 8KB, 16KB, 32KB$.

- ◇ For each $MEMORYPOOL\_SIZE$ value, use the same paper-and-pencil analysis to obtain the hit/miss rate. Put the results in a csv file named cache_capacity_analytical.csv with the format:
  `mempool_size,num_requests,num_hits,num_misses,hit_rate,execution_time`
- ◇ For each $MEMORYPOOL_SIZE$ value, name the C source file accordingly and run the simulation with the same configuration file as in the past section.
- ◇ For each simulation, put the results in a csv file named cache_capacity_experimental.csv with the same format as above.

8 Associativity

For this part, we modify the code snippet to be as follows:

```
#define KB 1024
#define MEMORYPOOL_SIZE 16*KB
#define STRIDE 8*KB
#define NUM_ITERATIONS 1

 for (j =0; j<NUM_ITERATIONS;j++)
 for (i = 0; i < (MEMORYPOOL_SIZE/2)/4; i+=1) {
    sum+=mempool[i];
    sum+=mempool[i+STRIDE/4];

}
```

- ◇ Assume same cache setup as in Section 4.1 and use the same paper-and-pencil analysis to obtain the hit/miss rate. except now, you will experiment with different cache organizations by sweeping the number of ways as follows (direct mapped cache, 4-ways, 16-ways, and 128-way) Put the results in a csv file named associativity_analytical.csv with the format:
  `way,num_requests,num_hits,num_misses,hit_rate,execution_time`

- ◇ For each *way* value, set the xml file accordingly (this is the nways parameter for L1), name the file accordingly and run the simulation with the C source file that implements the above configuration (you can modify the cache_test_2Arrays_4KB_4_1.c file or use it as reference to write your own).
- ◇ For each simulation, put the results in a csv file named associative_experimental.csv with the same format as above.

For all the parts, you need to comment on the comparison between your analytical results and the simulations ones.