CS 211 Spring 2017

Prof. Santosh Nagarakatte

Programming Assignment 4: Simulate Caches
Due: 5.00 PM, Friday April 21

Overview

The goal of this assignment is to provide you a better understanding of caches. You are required to write a cache simulator using the C programming languages. **The programs have to run on iLab machines and should be tested with the autograder.**

We are providing real program memory traces as input to your cache simulator**.** The format and structure of the memory traces are described below.

Memory Access Traces

The input to the cache simulator is a memory access trace, which we have generated by executing real programs. The trace contains memory addresses accessed during program execution. Your cache simulator will have to use these addresses to determine if the access is a hit, miss, and the actions to perform.

The memory trace file consists of multiple lines. Each line of the trace file corresponds to a memory accesses performed by the program. Each line consists of multiple columns, which are space-separated.The first column reports the PC (program counter) when this particular memory access occurred. Second column lists whether the memory access is a read (R) or a write operation. And the last column reports the actual 48-bit memory address that has been accessed by the program. In this assignment, you only need to consider the second and the third columns (i.e. you don't really need to know the PCs).

Cache Simulator

You will implement a cache simulator to evaluate different configurations of caches. It should be able to run with different traces files. The followings are the requirements for your cache simulator:

1. Simulate only **one level** cache: L1
2. The cache size, associativity, and block size are input parameters. Cache size and block size are specified in bytes.

3. Replacement algorithm: ***First In First Out*** **(FIFO)** .When a block needs to be replaced, the cache evicts the block that was accessed first. It does not take into account whether the block is frequently or recently accessed..
4. It's a **write through** cache.
5. You need to implement two types of caches as it will be explained later.

Running your Cache Simulator

You have to name your cache simulator **first.** Your program should support the following usage interface:

./first <cache size> <associativity> <block size> <trace file>

where:

A) < *cachesize* > is the total size of the cache in bytes. This number will be a **power of 2**.

B) < *associativity* > is one of:
- **direct** - simulate a direct mapped cache.
- **assoc** - simulate a fully associative cache.
- **assoc:n** - simulate an *n – way* associative cache. *n* will be a power of 2.

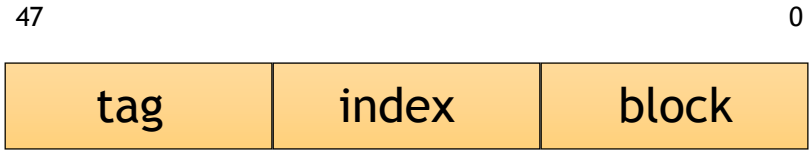
C) < *blocksize* > is a power of 2 integer that specifies the size of the cache block in bytes.

D) < *tracefile* > is the name of the trace file.

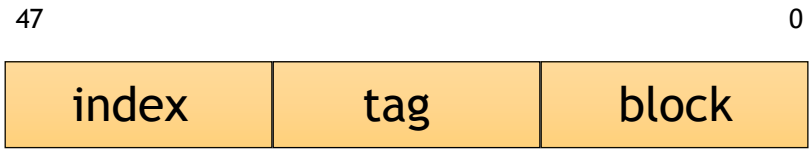
Note: Your program should give "reasonable" warning if the input is **not** in the right format.

In this assignment, you have to implement two types of caches (TypeA and TypeB). These caches differ in the bits of the address that they use to index into the cache.

Type A: These cache use the indexing schemes discussed in class. We identify whether the access is a hit or a miss using the tag, index, and block bits from the address as shown below.



Type B*: TypeB caches interprets the bits from the memory address differently. With Type B caches, the most significant bits of the 48-bit address constitute the index bits, which is followed by the tag bits and the block bits as shown below.*



Output of the program

Your program should print out the number of memory reads, memory writes, cache hits, and cache misses for each type of cache as shown below. **You should follow** the exact same format shown below, otherwise, the autograder can not grade your program properly.

```
$./first 32 assoc:2  4 trace1.txt
cache A
Memory reads: 173
Memory writes: 334
Cache hits: 827
Cache misses: 173
cache B
Memory reads: 667
Memory writes: 334
Cache hits: 333
Cache misses: 667
```

In this example above, we are simulating 2-way set associate cache of size 32 bytes. Each cache block is 4 bytes. The trace file name is "trace4.txt". As you can see, the simulator should simulate both catch types **simultaneously** (in the same run) and produce results (as shown above) for both cache types together.

Simulation Details

1. (a) When your program starts, there is nothing in the cache. So, all cache lines are empty. (b) you can assume that the memory size is 2^{48} . Therefore, memory addresses are **48 bit** (zero extend the addresses in the trace file if they're less than 48-bit in length). (c) the number of bits in the tag, cache address, and byte address are determined by the cache size and the block size; (d) Your simulator should simulate the operation of a cache according to the given parameters for the given trace

2. For a write-through cache, there is the question of what should happen in case of a write miss. In this assignment, the assumption is that the block is first read from memory (one read memory), and then followed by a memory write.

3- You do **not** need to simulate the memory in this assignment. Because, the traces doesn't contain any information on "data values" transferred between the memory and the caches.

4. You have to compile your program with the following flags: **-Wall -Werror -fsanitize=address**

Submission

You have to e-submit the assignment using **Sakai**. Put all files (**source code + Makefile + Report)** into a **directory** named **first,** which itself is a sub-directory under **pa4**. Then, create a tar file (follow the instructions in the previous assignments to create the tar file). Your submission should be **only** a **tar** file named pa4.tar. You must submit.

You have to e-submit the assignment using Sakai. Your submission should be a tar file named pa4.tar. To create this file, put everything that you are submitting into a directory named pa4. Then, cd into the directory containing pa4 (that is, pa4's parent directory) and run the following command:

tar cvf pa4.tar pa4

To check that you have correctly created the tar file, you should copy it (pa4.tar) into an empty directory

and run the following command:

tar xvf pa4.tar

This should create a directory named pa4 in the (previously) empty directory. Your pa4 folder should contain the following:

• readme.pdf: This file should briefly describe the main data structures being used in your program. You should also report your observation on which type of cache (A or B) gives better cache hit ratio ? and why?

• Makefile: There should be at least two rules in your Makefile:

1. first: build the executables (first).
2. clean: prepare for rebuilding from scratch.

• source code: all source code files necessary for building your programs. Your code should contain

at least two files: first.c and first.h.

Autograder

First mode

Testing when you are writing code with a pa4 folder

1. Lets say you have a pa4 folder with the directory structure as described in the assignment.
2. Copy the folder to the directory of the autograder
3. Run the autograder with the following command

$python auto_grader.py

It will run the test cases and print your scores.

Second mode

This mode is to test your final submission (i.e, pa4.tar)

1. Copy pa4.tar to the autograder directory
2. Run the autograder with pa4.tar as the argument as below:

$python auto_grader.py pa4.tar

Grading guidelines

The grading will be done in two phases, an automated phase and a manual phase.

Automated grading phase

This phase will be based on programmatic checking of your program. We will build a binary using the Makefile and source code that you submit, and then test the binary for correct functionality and efficiency against a set of inputs. Correct functionality and efficiency include, but are not limited to, the following:

1. We should be able build your program by just running make.

2. Your program should follow the format specified above for the usage interface.

3. Your program should strictly follow the input and output specifications mentioned above. (Note: This is perhaps the most important guideline: failing to follow it might result in you losing all or most of your points for this assignment. Make sure your program's output format is exactly as specified. Any deviation will cause the automated grader to mark your output as "incorrect". REQUESTS FOR RE-EVALUATIONS OF PROGRAMS REJECTED DUE TO IMPROPER FORMAT WILL NOT BE ENTERTAINED.)

Manual grading phase

This phase will involve manual inspection of your code and the readme.pdf file provided by you along with the code. Note that **only** programs that pass the automate phase test cases ("to some extent") will be eligible for grading in the manual phase.