CS211 Spring 2018
Programming Assignment V

David Menendez

Due: May 1, 2018, at 5:00 AM

This assignment will provide a better understanding of caches. It has two parts. In the first part (80 points), you will write a program that simulates cache behavior given a configuration and a memory access trace. In the second part (20 points), you will describe how you designed your simulator and observe how the configuration choices affect memory usage.

1 Cache Simulation

A *cache* is a collection of *cache lines*, each of which may store a *block* of memory along with some additional information about the block (for example, which addresses it contains). Each block contains the same number of bytes, known as the *block size*. The block size will always be a power of two. The *cache size* is the block size multiplied by the number of cache lines (that is, the additional information is not counted in the cache size).

Consider a system with 48-bit addresses and a block size of 16 bytes. Each block will begin with an address divisible by 16. Because $16 = 2^4$, the last 4 bits of an address will determine its *offset* within a particular block. For example, the address `ffff0000abcd` (hexadecimal) will have offset `d`. The remaining 44 bits of the address may be considered a *block identifier*.

If the block size were 8 bytes, then the last 3 bits of the address will be its block offset. The last three bits of `ffff000abcd` are `101` (binary) or `5` (hexadecimal). The most-significant 45 bits will be the block identifier. (Exercise: write the block identifier in hexadecimal.[1])

For a cache with a single cache line, that cache line will store the 16 bytes of the block along with the block identifier. Each memory access will first check whether the address is part of the block currently in the cache (if any).

1.1 Memory operations

Your simulator will simulate two memory operations: reading and writing individual bytes. Your program will read a trace file describing addresses to read or write from, and will keep track of which blocks are stored in which cache lines in order to determine when these memory operations result in actual reads and writes to memory.

Note that your program only keeps enough information to simulate the behavior of the cache. It does not need to know the actual contents of memory and the blocks stored in the cache lines; this information is, in fact, not available.

[1] `1ffe0001579`

Your program will simulate a *write-through* cache. The operations supported are reading and writing from an address A.

read A Read the byte at address A. If the block containing A is already loaded in the cache, this is a *cache hit*. Otherwise, this is a *cache miss*: note a *memory read* and load the block into a chosen cache line (see sections 1.2 and 1.3).

write A Write a byte at address A. If the block containing A is already loaded in the cache, this is a cache hit: note a *memory write*. Otherwise, this is a cache miss: note a memory read, load the block into a chosen cache line, and then note a memory write.

Your program will track the number of cache hits, cache misses, memory reads, and memory writes.

Note that loading an address or block into the cache means changing the information about a particular cache line to indicate that it holds a particular block. Since your simulator does not simulate the contents of memory, no data will be loaded or stored from the address itself.

1.1.1 Prefetching

Prefetching is a technique used to increase spatial locality in a cache. In the operations described above, blocks are read from memory only after a cache miss. If a cache uses prefetching, each cache miss will also ensure that the next block is loaded into the cache.

That is, if an attempt to access A results to a cache miss, the simulator will load the block containing A into the cache. A prefetching cache will also check whether the block containing $A + B$ (where B is the block size) is present in the cache and will load it if not.

This pseudo-code illustrates how to simulate a memory read when prefetching:

```
read(A):
    if A in cache:
        increment cache_hits
    else:
        increment cache_misses
        increment memory_reads
        load A
        if A+block_size not in cache:
            increment memory_reads
            load A+block_size
```

Note that the prefetching step is not considered a cache hit or a cache miss.

The behavior is similar for write operations, except that the prefetched block is not written back to memory.

Your program will simulate the behavior of a cache with and without prefetching, and will give the number of cache hits, cache misses, memory reads, and memory writes for both scenarios.

1.2 Mapping

For caches with multiple cache lines, there are several ways of determining which cache lines will store which blocks. Generally, the cache lines will be grouped into one or more *sets*. Whenever a

block is loaded into a cache line, it will always be stored in a cache line belonging to its set. The number of sets will always be a power of two.

Let B be the block size. The least significant $\log_2 B = b$ bits of an address will be its block offset. If there are S sets, then the next least significant $\log_2 S = s$ bits of the address will identify its set. The remaining $48 - s - b$ bits are known as the *tag* and are used to identify the block stored in a particular cache line.

For example, for a cache with 16 sets and 16-byte blocks, the address `ffff0000abcd` is in set 12 (`c`). The first 40 bits, `ffff000ab`, are the tag.

Your program will simulate three forms of cache:

Direct-mapped This is the simplest form of cache, where each set contains one cache line and no decisions need to be made about where a particular block will be stored.

n-way associative In this form of cache, each set contains n cache lines, where n is a power of two. The particular block stored in each line is identified by its tag.

Fully associative This form of cache puts all the cache lines in a single set.

For direct and n-way associative caches, your program will need to derive the number of sets (S) from the *associativity* A (the number of cache lines per set), the block size B, and the size of the cache C using the relation $C = SAB$. For fully associative caches, $S = 1$, but you will need to determine A using the same relation.[2]

1.3 Replacement policies

Each cache line is either valid, meaning it contains a copy of a block from memory, or invalid, meaning it does not. Initially, all cache lines are invalid. As cache misses occur and blocks are loaded into the cache, those lines will become valid. Eventually, the cache will need to load a block into a set which has no invalid lines. To load this new block, it will select a line in the set according to its *replacement policy* and put the new block in that line, replacing the block that was already present.[3]

We consider two replacement policies in this assignment:

fifo ("First-in, first-out") In this policy, the cache line that was loaded least recently will be replaced.

lru ("Least-recently used") In this policy, the cache line that was accessed least recently will be replaced.[4]

Your simulator will implement fifo. Implementing lru is left for extra credit.

2 Program (80 points)

You will write a program `cachesim` that reads a trace of memory accesses and simulates the behavior of two caches for that trace. The program will be awarded up to 60 points by the auto-grader. The

[2] Can direct-mapped and fully associative caches be considered special cases of n-way associativity?

[3] For direct-mapped caches, no decision needs to be made, because each set only contains one cache line.

[4] For a prefetching cache, the prefetched block is considered to be accessed if it is loaded from memory, but is *not* accessed if it is already in the cache.

remaining 20 points will be granted for style. An optional second part may be completed for extra credit.

Your program will take five arguments. These are:

1. The *cache size*, in bytes. This will be a power of two.
2. The *associativity*, which will be "direct" for a direct-mapped cache, "assoc:n" for an n-way associative cache, or "assoc" for a fully associative cache (see section 1.2)
3. The *replacement policy*, which will be "fifo" or "lru" (see section 1.3).
4. The *block size*, in bytes. This will be a power of two.
5. The *trace file*.

Your program will use the first four arguments to configure two caches, one which prefetches and one which does not. It will read the memory accesses in the trace file and simulate the behavior of both caches. When it is complete, it will print the number of cache hits, cache misses, memory reads, and memory writes for both caches.

Usage

```
./cachesim 512 direct fifo 8 trace1.txt
```

Your program SHOULD confirm that the block size and cache size are powers of two. If the associativity is "assoc:n", your program SHOULD confirm that n is a power of two.

Your program MUST implement the "fifo" replacement policy. Your program MAY implement "lru" for extra credit.

Input The input is a memory access trace produced by excuting real programs. Each line describes a memory access, which may be a read or write operation. The lines contain three columns, separated by spaces. The first is the program counter (PC) at the time of the access, followed by a colon. The second is "R" or "W", indicating a read or write, respectively. The third is the 48-bit memory address which was accessed. Additionally, the last line of the file contains only the string "#eof".

Here is a sample trace file:

```
0x804ae19: R 0x9cb3d40
0x804ae19: W 0x9cb3d40
0x804ae1c: R 0x9cb3d44
0x804ae1c: W 0x9cb3d44
0x804ae10: R 0xbf8ef498
#eof
```

You MAY assume that the trace file is correctly formatted.

Output Your program will print the number of cache hits, cache misses, memory reads, and memory writes observed when simulating the memory access trace, both with no prefetching and with prefetching.

The output must have this form:

```
no-prefetch
Cache hits: 6501
Cache misses: 3499
Memory reads: 3499
Memory writes: 2861
with-prefetch
Cache hits: 8124
Cache misses: 1876
Memory reads: 3521
Memory writes: 2861
```

Note that spacing and capitalization must be correct for the auto-grader to award points. **We will not give partial credit when points are lost due to improper formatting.**

3 Report (20 points)

In addition to writing the simulator program, you will submit a brief report describing your developement of the program and answering a few questions. Your report should describe the data structure(s) used to represent the cache and the algorithms used to simulate memory accesses. In addition, describe your testing strategy and how you determined that your simulator is correct.

Your report should then briefly answer these questions:

1. How does the prefetcher affect the number of cache hits and memory reads? Is this effect consistent across all testing? Why or why not?
2. Would it be possible to adapt your simulator to simulate a two-level cache (that is, to simulate an L1 and L2 cache)? What parts of your program would need to change?
3. How would you modify your program to simulate a cache that does not immedately write changed blocks back to memory? What would need to change?

Please be clear and concise in your answers, but provide enough detail for the grader to understand your points. Aim to provide between one paragraph and two pages for each answer.

Acceptable formats for your report are plain text, HTML, and PDF. PDF is recommended. Reports in other formats, such as Microsoft Word, will not be graded. If you submit as HTML, your file must be readable in a browser without an Internet connection.

4 Submission

Your solution to the assignment will be submitted through Sakai. You will submit a Tar archive file containing your circuit designs, and the source code and makefile for your program. Your archive should not include any compiled code or object files.

The remainder of this section describes the directory structure, the requirements for your makefile, how to create the archive, and how to use the provided auto-grader.

4.1 Directory structure

Your project should be stored in a directory named `src`. This directory will contain (1) a makefile, (2) any source files needed to compile `cachesim`, and (3) your report.

This diagram shows the layout of a typical project:

```
src
+- Makefile
+- cachesim.c
+- report.pdf
```

If you are using the auto-grader to check your program, it is easiest to create the `src` directory inside the `pa5` directory created when unpacking the auto-grader archive (see section 4.4).

4.2 Makefiles

We will use `make` to manage compilation. Each program directory will contain a file named `Makefile` that describes at least two targets. The first target (invoked when calling `make` with no arguments), should compile the program. An additional target, `clean`, should delete any files created when compiling the program (typically just the compiled program).

You may create this makefile using a text editor of your choice.

A typical makefile would be:

```
cachesim: cachesim.c
	gcc -g -Wall -Werror -fsanitize=address -o cachesim cachesim.c

clean:
	rm -f cachesim
```

Note that the command for compiling `cachesim` uses GCC warnings and the address sanitizer. You will lose one style point if your makefile does not include these.

You are not required to use `-g`. Including it will enable debugging using `gdb` and may improve AddressSanitizer error messages.

Note that the makefile format requires tab characters on the indented lines. Text copied from this document may not be formatted correctly.

You are strongly encouraged to use `make` when compiling your code. In addition to making sure you receive warnings as early as possible, it will save you the time needed to manually invoke `gcc`.

4.3 Creating the archive

We will use `tar` to create the archive file. To create the archive, first ensure that your `src` directory contains only the source code and makefile needed to compile your project and the report. Any compiled programs, object files, or other additional files must be moved or removed.

Next, move to the directory containing `src` and execute this command:

```
tar czvf pa5.tar src
```

`tar` will create a file `pa5.tar` that contains all files in the directory `src`. This file can now be submitted through Sakai.

To verify that the archive contains the necessary files, you can print a list of the files contained in the archive with this command:

```
tar tf pa5.tar
```

On some operating systems, `tar` may find or create hidden files and include them in your archive. This is usually not a problem.

4.4 Using the auto-grader

We have provided a tool for checking the correctness of your project. The auto-grader will compile your programs and execute them several times with different arguments, comparing the results against the expected results.

Setup The auto-grader is distributed as an archive file `pa5_grader.tar`. To unpack the archive, move the archive to a directory and use this command:

```
tar xf pa5_grader.tar
```

This will create a directory `pa5` containing the auto-grader itself, `grader.py`, a library `autograde.py`, and a directory of test cases `data`.

Do not modify any of the files provided by the auto-grader. Doing so may prevent the auto-grader from correctly assessing your program.

You may create your `src` directory inside `pa5`. If you prefer to create `src` outside the `pa5` directory, you will need to provide a path to `grader.py` when invoking the auto-grader.

Usage While in the same directory as `grader.py`, use this command:

```
python grader.py
```

The auto-grader will compile and execute the program `cachesim` in a directory `src` contained in the current working directory.

By default, the auto-grader will grade the project including extra credit. To grade only a particular part, give its name as an argument. For example:

```
python grader.py cachesim:fifo
```

To obtain usage information, use the `-h` option.

Program output By default, the auto-grader will not print the output from your program, except for lines that are incorrect. To see all program output for unsuccessful tests, use the `-v` option:

```
python grader.py -v
```

To see program output for all tests, use `-vv`. To see no program output, use the `-q` option.

Checking your archive You SHOULD use the auto-grader to check an archive before (or just after) submitting. To do this, use the `-a` option with the archive file name. For example,

```
python grader.py -a pa5.tar
```

This will unpack the archive into a temporary directory, grade the programs, and then delete the temporary directory.

Specifying source directory If your `src` directory is not located in the same directory as `grader.py`, you may specify it using the `-s` option. For example,

```
python grader.py -s ../path/to/src
```

Note: before testing your program, the auto-grader will execute `make clean` and `make` in the program directory. If either command fails for any reason, such as compiler errors or a missing or invalid makefile, you will receive no points for the program. Before submitting, make sure that the auto-grader can successfully comple and test your project on an iLab machine.

5 Grading

This assignment is worth 100 points. The program described in section 2 is worth 80 points, of which 60 will be determined by automated testing and 20 will be determined by examination of source code. The remaining 20 points will be granted based on your report. Your program MUST successfully compile and execute when using the auto-grader on an iLab machine in order to receive points. The auto-grader is provided so that you can confirm that your submission can be tested successfully. **It is your responsibility to ensure that your program can be tested by the auto-grader.**

Make sure that your programs meet the specifications given, even if no test case explicitly checks it. It is advisable to perform additional tests of your own devising, which you may describe in the testing strategy section of your report.

5.1 Style points

Your program will receive up to 20 style points. These will be awarded by graders as part of a manual inspection of your code. The graders will consider several aspects of your program, such as organization and use of comments.

Some advice for winning points:

- Make sure to use `-Wall -Werror -fsanitize=address` when compiling.
- Use indentation to make your code more readable.
- Document your code with comments. At a minimum, describe the purpose of any functions you create.
- Do not put all your code in a single function.

This program can be broken into two layers: an inner layer that simulates the cache operations, and an outer layer that reads the input and generates output. Being aware of these layers may help you organize your code into functions.

5.2 Academic integrity

You must submit your own work. You should not copy or even see code for this project written by anyone else, nor should you look at code written for other classes. We will be using state of the art plagiarism detectors. Projects which the detectors deem similar will be reported to the Office of Student Conduct.

Do not post your code on-line or anywhere publically readable. If another student copies your code and submits it, both of you will be reported.

6 Advice

Start working early. Before you do any coding, make sure you can run the auto-grader, create a Tar archive, and submit that archive to Sakai. You don't want to discover on the last day that `scp` doesn't work on your personal machine.

Decide your data structures and algorithms first. Writing out pseudocode is not required, but it may be a good idea.

Start working early. You will almost certainly encounter problems you did not anticipate while writing this project. It is much better if you find them in the first week and can ask questions.