CS211 Fall 2018
Programming Assignment V

David Menendez

Due December 14, 2018
Submit to Sakai by December 15, 3:00 AM

This assignment will provide a better understanding of caches and provide further experience with C programming.

Advice Read this document first. Take notes. In particular, read section 1.3 carefully.

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.

Do not panic. I am aware that there is only a week left in the semester and that we all have final projects and other end-of-semester responsibilities. I will take this into account when determining the final grades. Begin with the simple cases (e.g., direct-mapped caches), and complete as much of the funcionality as you are able.

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])

[1] `1ffe0001579`

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). Since we are only simulating memory reads and writes, you cache will not need to store the actual blocks.

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.

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 1.4).

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):
    B <- block id for A
    if B in cache:
        increment cache_hits
    else:
        increment cache_misses
        increment memory_reads
```

```
store B in cache
C <- block id for A + block_size
if C not in cache:
    increment memory_reads
    store C in cache
```

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 or without prefetching.

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 Calculating block, set, and tag

We will simulate a cache in a system with 48-bit addresses. Since `int` has 32 bits on the iLab machines, you will want to use a `long` or `unsigned long` to represent the addresses. Using some other type, such as a string containing hexadecimal digits, is not recommended for efficiency reasons.

Your program will need to extract sequences of bits from the addresses it works with. When simulating a cache with block size $B = 2^b$, the least significant b bits of an address (the block offset) may be ignored. Code such as $X >> b$ will effectively remove the block offset, leaving only the block identifier.

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

The least significant s bits of the block identifier identify the set. To obtain these, recall that $2^k - 1$, represented in binary, is all zeros except for the last k bits, which are ones. That is, $2^2 - 1$ is 11_2, $2^4 - 1$ is 1111_2, and so forth. Recall that 2^k may be easily computed by $1 << k$.

The tag is the portion of the address remaining after removing the block offset and set identifier.

1.4 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.

Note that each set contains a fixed, finite number of cache lines. To determine the oldest block in the cache, it is sufficient to store the *relative* age of each cache line. That is, the first time a block is loaded into a cache line, its relative age is set to 0 and the ages of all other valid cache lines are increased by 1. This ensures that every cache line will have a unique age. Once all cache lines are valid (i.e., contain data), a sequential search is sufficient to determine the oldest cache line.[5]

2 Program

You will write a program `cachesim` that reads a trace of memory accesses and simulates the behavior of a cache for that trace. The program will be awarded up to 300 points by the auto-grader. (The version of the auto-grader included for your testing gives up to 150 points.) An optional second part may be completed for extra credit.

Your program will take six 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 *prefetching policy*, which will be "p0" (no prefetching) or "p1" (prefetch one block ahead).
4. The *replacement policy*, which will be "fifo" or "lru" (see section 1.4).
5. The *block size*, in bytes. This will be a power of two.

[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.

[5] Sequential search is $O(n)$, but for our purposes n will always be fairly small.

6. The *trace file*.

Your program will use the first four arguments to configure a cache. It will read the memory accesses in the trace file and simulate the behavior of that cache. When it is complete, it will print the number of cache hits, cache misses, memory reads, and memory writes.

Usage

```
./cachesim 512 direct p0 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 fields, 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. Note that the PC is not used by your simulator, and may be ignored.

Recall that the format code for hexadecimal integers is `x`. To read and discard an arbitrarily large hex int, you may use `%*x`. To read a hex int and store in a `long int`, you should use `%lx`.

Output Your program will print the number of cache hits, cache misses, memory reads, and memory writes observed when simulating the memory access trace.

The output must have this form:

```
Memory reads: 3499
Memory writes: 2861
Cache hits: 6501
Cache misses: 3499
```

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 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.

3.1 Directory structure

Your project should be stored in a directory named `src`. This directory will contain a makefile and any source files needed to compile `cachesim`.

This diagram shows the layout of a typical project:

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

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 3.4).

3.2 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. Your score will be reduced by 20% 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`.

3.3 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. 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.

If is also possible explicitly include files when creating the Tar archive. For example

```
tar czvf pa5.tar src/Makefile src/cachesim.c
```

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.

3.4 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 not grade the project including extra credit. Use the `-x` or `--extra` options to enable testing of extra credit.

To grade only a particular part, give its name as an argument. For example:

```
python grader.py -x cachesim:fifo.p0 cachesim:lru.p0
```

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.

4 Grading

This assignment is worth 300 points. 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.

4.1 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.