CS211 Fall 2021
Programming Assignment V

David Menendez

Due December 13, 2021
Submit by December 15, 11:59 PM

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.

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 would 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 (the data) along with the validity bit and block identifier (the metadata). 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] `1ffe0001579`

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, write-allocate* cache. The operations supported are reading and writing from an address A.

read A Simulate reading 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 Simulate writing 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 the benefit of spatial locality for 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 ensure that the next block is *also* loaded into the cache.

For example, a program accesses address A, which is part of block X. If this access is a cache miss (meaning X is not in the cache), the cache will load block X into the cache. A prefetching cache will also check whether block $X + 1$ is present in the cache, and load it into the cache if not. Note that the prefetching step is not considered a cache hit or a cache miss.

Algorithm 1 illustrates how to simulate a memory read when prefetching.

Prefetching occurs when loading blocks into the cache. Because `cachesim` simulates a write-allocate cache, a cache miss during a write will result in loading the relevant block, which may then result in prefetching the next block. Note that prefetching does not increase or decrease the number of writes.

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.

Algorithm 1 Simulating a memory read in a prefetching cache

```
procedure READ(A)
    X ← block ID for A
    if X in cache then
        increment cache hits
    else
        increment cache misses
        increment memory reads
        set X in cache
        Y ← X + 1
        if Y not in cache then
            increment memory reads
            set Y in cache
        end if
    end if
end procedure
```

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)

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

may be ignored. Code such as $X >> b$ will effectively remove the block offset, leaving only the block identifier.

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.

Note Addresses are integers, which are bit vectors. Extracting the set index and tag from an address SHOULD be done using the bitwise operations provided by C. Your program SHOULD NOT use functions from `math.h` or attempt to manipulate addresses as strings. Integer operations are fast. Use them!

Exercise Show that the following expression is true if and only if `n` is a power of two:

```
x != 0 && (x & (x - 1)) == 0
```

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]

when there's no set with invalid lines, follow the replacement policy

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, with and without prefetching. The program will be awarded up to 100

[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.

points by the auto-grader. (The version of the auto-grader included for your testing gives up to 50 points.) 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 1.4).
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 simulated caches: one that prefetches and one that does not. It will read the memory accesses in the trace file and simulate the behavior of those caches. When it is complete, it will print the number of cache hits, cache misses, memory reads, and memory writes observed for both simulated 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 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, both for a cache without prefetching and with prefetching.

The output must have this form:

```
Prefetch 0
Memory reads: 168
Memory writes: 334
Cache hits: 832
Cache misses: 168
Prefetch 1
Memory reads: 345
Memory writes: 334
Cache hits: 827
Cache misses: 173
```

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 Example

Debugging `cachesim` can be a challenge without small examples with known behavior. It is not a bad idea to create several small trace files to exercise the behavior of your simulator. In particular, you will want to be sure that your code correctly identifies the tag and set for each memory access, that it correctly detects whether the requested memory block is present in the cache, that it correctly loads in response to a cache miss, and that it correctly follows the cache replacement policy.

For example, consider a 128-byte cache with 16-byte blocks and 2-way associativity. As $128 = 8 \times 16$, this cache will have 8 cache lines divided into 4 sets. Thus, the block offset will be 4 bits and the set ID will be 2 bits. With this cache configuration, the memory accesses described in `trace0.txt` will involve six tags, which we will abbreviate for brevity.

The following chart illustrates the behavior of a regular (non-prefetching) cache. For each memory access, we give the tag abbreviation and set number, whether it was a hit or miss, and the tags in each set in the order they were added.

```
       Address          Tag Set            0    1    2     3
0x1: R 0xffff00000000   X   0       MISS   X,-  -    -     -
0x2: R 0xffff00000008   X   0       HIT    X,-  -    -     -
0x3: R 0xffff00000010   X   1       MISS   X,-  X,-  -     -
0x4: R 0xffff00000018   X   1       HIT    X,-  X,-  -     -
0x5: R 0xffff00000020   X   2       MISS   X,-  X,-  X,-   -
0x6: R 0xffff00000030   X   3       MISS   X,-  X,-  X,-  X,-
0x7: R 0xffff00000040   Y   0       MISS   Y,X  X,-  X,-  X,-
0x8: R 0xffff00000000   X   0       HIT    Y,X  X,-  X,-  X,-
0x9: R 0xffff00000080   Z   0       MISS   Z,Y  X,-  X,-  X,-
0xa: R 0xffff00000040   Y   0       HIT    Z,Y  X,-  X,-  X,-
0xb: R 0x000000000030   T   3       MISS   Z,Y  X,-  X,-  T,X
0xc: R 0x000000000080   V   0       MISS   V,Z  X,-  X,-  T,X
```

```
0xd: R 0x000000000040   U   0       MISS    U,V  X,-  X,-  T,X
0xe: R 0xffff00000044   Y   0       MISS    Y,U  X,-  X,-  T,X
0xf: R 0x000000000040   U   0       HIT     Y,U  X,-  X,-  T,X
```

Overall, this is 5 hits, 10 misses, and 10 reads from memory.

Compare this with the behavior of a prefetching cache:

```
       Address          Tag Set            0    1    2    3
0x1: R 0xffff00000000   X   0       MISS    X,-  X,-  -    -
0x2: R 0xffff00000008   X   0       HIT     X,-  X,-  -    -
0x3: R 0xffff00000010   X   1      *HIT     X,-  X,-  -    -
0x4: R 0xffff00000018   X   1       HIT     X,-  X,-  -    -
0x5: R 0xffff00000020   X   2       MISS    X,-  X,-  X,-  X,-
0x6: R 0xffff00000030   X   3      *HIT     X,-  X,-  X,-  X,-
0x7: R 0xffff00000040   Y   0       MISS    Y,X  Y,X  X,-  X,-
0x8: R 0xffff00000000   X   0       HIT     Y,X  Y,X  X,-  X,-
0x9: R 0xffff00000080   Z   0       MISS    Z,Y  Z,Y  X,-  X,-
0xa: R 0xffff00000040   Y   0       HIT     Z,Y  Z,Y  X,-  X,-
0xb: R 0x000000000030   T   3       MISS    U,Z  Z,Y  X,-  T,X
0xc: R 0x000000000080   V   0       MISS    V,U  V,Z  X,-  T,X
0xd: R 0x000000000040   U   0      *HIT     V,U  V,Z  X,-  T,X
0xe: R 0xffff00000044   Y   0       MISS    Y,V  Y,V  X,-  T,X
0xf: R 0x000000000040   U   0      *MISS    U,Y  U,Y  X,-  T,X
```

Asterisks mark changes from the non-prefetching version. In particular, note line `0xb`, where reading block `0x00000000003` (T3) prefetches block `0x00000000004` (U0). This results in blocks `0x00000000004` and `0x00000000008` (V0) being loaded in the opposite order, changing the last access from a hit to a miss.

Overall, this is 7 hits, 8 misses, and 16 reads from memory.

Exercise: classify each miss as cold, conflict, or capacity.

Exercise: work out how a similarly configured LRU cache would behave for this memory access trace.

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 Provided files

The archive |pa5-grader.tar| includes the auto-grader script and a makefile that you may use to manage your testing and submission process. (Note that this makefile is separate from the makefile used to compile your program.)

To extract the files, download the archive to an iLab machine and use this command:

```
$ tar -xf pa5-grader.tar
```

This will create a directory `pa5/` with this structure:

```
pa5
+- autograde.py
+- data/
+- grader.py
+- template.mk
```

4.2 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 4.5).

You are free to use multiple files with whatever names you like, as long as your Makefile can create your `cachesim` program. (C source code files should have extensions `.c` and `.h`, as appropriate.)

You MAY use the provided `template.mk` as-is for your makefile, by moving or copying it into `src/` and renaming it. E.g.,

```
pa5$ mv template.mk src/Makefile
```

4.3 Makefiles

Your `src/` directory (section 4.2) must contain a makefile describing how to compile your program. The auto-grader will use the command `make` and expect this to compile your program. You are strongly encouraged to use make to compile your program during development and testing, as this will reduce confusion caused by using inconsistent compiler settings.

Your makefile must describe two targets. In addition to `make cachesim`, your makefile should define `make clean`, which will delete `cachesim` and any other files created during compilation.

```
TARGET = cachesim
SRC    = $(TARGET).c
CC     = gcc
CFLAGS = -g -Wall -Wvla -Werror -fsanitize=address,undefined

$(TARGET): $(SRC)
        $(CC) $(CFLAGS) -o $@ $^

clean:
        rm -rf $(TARGET) *.o *.a *.dylib *.dSYM
```

Note that this makefile specifies several compiler options when compiling `cachesim`. If you choose to make your own makefile, you must compile using at least the options shown here.

Note also that `cachesim` is the first target defined, meaning it can be compiled simply typing `make` will with no arguments.

You may extend this makefile with additional targets for your own use. For example, if you are frequently testing your program with a particular test file, you might add a target like this:

```
test: cachesim
        ./cachesim my-test-file.txt
```

With this definition, the command `make test` will compile your program and then run it with your test argument.

Warning If you copy and paste make code out of this document, you must replace the initial indentation with tabs. Make requires tabs when giving the commands associated with a target.

4.4 Creating the archive

We will use `tar` to create the archive file. You may use the makefile in `pa5/` to create the archive, or you may call `tar` directly.

From the directory containing `src/`, execute this command:

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

(If you have used additional files or used a different name for your source file, modify this command accordingly.)

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
```

You may also use the auto-grader to confirm that your archive is complete. This command will unpack the archive in a temporary directory, and then compile and test your program:

```
pa5$ ./grader.py -a pa5.tar
```

(The argument to `-a` can be a path, in case you did not create your archive in the same directory as the grader.)

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

4.5 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.

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

```
pa5$ ./grader.py
```

The auto-grader will compile and execute the program in the directory `src/`, assuming `src/` has the structure described in section 4.2. The compiled program will be in a directory `build/`, which is created by the auto-grader.

During development, you may prefer to use the `--stop` or `-1` option, which produces more program output but stops after the first failed test case.

```
pa5$ ./grader.py -1
```

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

```
pa5$ ./grader.py cachesim:fifo
```

To obtain usage information, use the `--help` or `-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 `--verbose` or `-v` option:

```
pa5$ ./grader.py -v
```

To see program output for all tests, use `-vv`. To see no program output, use the `-q` option. Note that `-1` implies `-v`.

Controlling LeakSanitizer Depending on your compiler, AddressSanitizer may activate LeakSanitizer (LSan) by default. LSan will check whether any memory is still allocated when your program terminates and report an error message. This will cause the auto-grader to report the test case as failed, even if your program provided the correct output.

To disable LSan, use the option `--lsan off`. To enable LSan, use the option `--lsan on`. Note that LSan is not be available on all platforms, but is available in iLab.

Leaking memory is still an error, and you will lose some points if your program leaks memory.

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,

```
pa5$ ./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,

```
pa5$ ./grader.py -s ../path/to/src
```

Note Before testing your program, the auto-grader will execute `make` using the makefile from your `src/` directory. If this 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. 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.

5.1 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.