Programming assignment 5 (120 points)

Start Assignment

Due Apr 26 by 11:59pm **Points** 120 **Submitting** a file upload **File Types** tar
Available until May 12 at 11:59pm

Simulating a cache and optimizing programs for caches

Prof. Yipeng Huang

Rutgers University

2021 Spring

0. Introduction and setup

Learning goals

You will see why caches are a vital component of modern computer architectures. You will experiment with different cache designs, and see how they impact the ability for caches to create an illusion of fast access from the CPU to main memory. Caches are invisible to the programmer—that is, programmers do not "program" caches, but you can still write your programs in a way that maximizes spatial and ▶ oral locality, so that caches can further improve program performance.

Resources

Get started early! You can post questions to the class Piazza (please avoid sending emails to the TAs and instructor).

- You can discuss the assignment with your classmates, but you cannot copy code from your classmates.
- You can use the internet to research the mechanics of the C programming language, how caches work, and how to write cache-friendly implementations of algorithms, but you cannot use websites that offer ready-made answers specific to this assignment and this class. Consider citing any sources

that you use in your research; such a citation can be as simple as providing a link to a webpage in a code comment.
- We will be checking the assignments for identical and/or plagiarized code using automated tools.
- It is not acceptable to share or post any course materials online without explicit permission from the instructor.
- **https://www.cs.rutgers.edu/academics/undergraduate/academic-integrity-policy (https://www.cs.rutgers.edu/academics/undergraduate/academic-integrity-policy)**
- **https://www.cs.rutgers.edu/academics/undergraduate/academic-integrity-policy/programming-assignments (https://www.cs.rutgers.edu/academics/undergraduate/academic-integrity-policy/programming-assignments)**
- The instruction staff takes the Rutgers University Academic Integrity Policy seriously. Please review it at: **https://nbprovost.rutgers.edu/academic-integrity-students (https://nbprovost.rutgers.edu/academic-integrity-students)** . We will report all violations.

Setup

Access the programming assignment files for this assignment by pulling new files from the class Github repository, 2021_0s_211:

```
git pull
```

If you do not have this directory already, clone the repository:

```
git clone https://github.com/yipenghuang0302/2021_0s_211.git
```

The files for this assignment are in the directory 2021_0s_211/pa5/.

---

1. FullyAssociative: Simulating a fully associative cache with FIFO cache replacement policy (easy) (24 points)

Your task in this section is to build a program that simulates the behavior of a CPU cache.

Input format

Your program should take a single command line argument specifying the path to an input file. The input file is a memory access trace that lists memory locations that some other example program had

accessed over the course of its execution. Example input files are in the tests/ directory. Each line in the input file lists a memory access, and contains the following information:

1. The first field is a sequence of characters " X ", which is a space, followed by a capital letter, followed by another space. The capital letter indicates what type of memory access took place. 'L' indicates a load, where the CPU reads data from the memory. 'S' indicates a store, where the CPU writes data to the memory. 'M' indicates a modify, which is essentially a load followed immediately by a store. In some memory trace files, you will see lines that start with "I ", which is an 'I', followed by two spaces; you can ignore these lines as they indicate memory accesses for instructions.
2. The second field is a hexadecimal number indicating the memory address that the CPU accessed, followed by a comma.
3. The third and final field is a decimal number indicating how many bytes were accessed from that memory location. For this assignment you can ignore this field since the data accesses are all smaller than the cache block size.
4. Note that the memory access traces only trace the addresses that were accessed. The actual memory contents that were transferred is not recorded.

The memory accesses are listed in the trace such that later accesses are listed later in the file.

Cache design parameters

For the part of the assignment, your cache simulator should have the following design parameters:

- The cache is an L1 data cache. L1 means that it is the fastest, smallest capacity cache closest to the CPU. It is a data cache, so any accesses to memory addresses that are program instructions are ignored.
- Placement policy: 16-way fully associative with 16-byte blocks; with the following parameters:
  - The number of set index bits s=0, such that the number of sets is S=1, thereby making this a fully associative cache.
  - The number of lines per set E=16.
  - The number of bits in the block offset b=4, such that the number of addresses in each block B=16.
  - The overall capacity of this cache is therefore 256 bytes.
- Replacement policy: First in, first out (FIFO). If the cache is full, then the cache line that was first allocated will be the next victim of eviction.
- Writing policy: write-back and write-allocate.

When the cache simulator starts up, you should assume that the cache is cold, that is, all lines are invalid and nothing is cached.

Output format

Expected outputs from your program for each test case are in the answers/ directory. You should print a single line formatted like so:

"hits:16 misses:1 evictions:0"

Indicating that memory accesses led to 16 cache hits, 1 cache miss, and no cache evictions.

How to compile, run, and test your code

- Instructions from Programming Assignment 1, 2, and 3 carry over.
- We have provided a compiled binary executable, pa5/csim-ref, which is a reference cache simulator. You can see the instructions on how to use it by running csim-ref with no arguments. It has a particularly useful "verbose" mode which will print whether each memory access resulted in a hit, miss, and/or eviction, in addition to the summary statistics that you are required to print. For example, to print the detailed simulation for the fullyAssociative cache on trace_0.txt, you can run from the pa5/ directory:

  ```
  ./csim-ref -v -s 0 -E 16 -b 4 -l 0 -t ./fullyAssociative/tests/trace_0.txt
  ```

- The autograder.py scripts for this assignment only work with Python version 3 on the ilab machines. This means that the correct way to invoke the autograder script is through either of these two commands:

  ```
  ./autograder.py
  ```

  or

  ```
  python3 autograder.py
  ```

---

2. directMapped: Simulating a direct-mapped cache (easy) (24 points)

Your task in this section is to modify your simulator so that it simulates a direct-mapped cache.

Input format

The input format for the previous part, fullyAssociative, carries over.

Cache design parameters

For the part of the assignment, your cache simulator should have the following design parameters:

- The cache is an L1 data cache.
- Placement policy: 256-byte direct-mapped cache with 16-byte blocks; with the following parameters:
  - The number of set index bits s=4, such that the number of sets is S=16. Given the choice of s and b, you can think about how to use the memory addresses to deduce which cache set each access must go to.
  - The number of lines per set E=1, making this a direct-mapped cache.
  - The number of bits in the block offset b=4, such that the number of addresses in each block B=16.
  - The overall capacity of this cache is therefore still 256 bytes.
- Replacement policy: None. This is a direct-mapped cache, which means that if a cache set line is already occupied, any cache misses will immediate evict the existing cache line.
- Writing policy: write-back and write-allocate.

When the cache simulator starts up, you should assume that the cache is cold, that is, all lines are invalid and nothing is cached.

Output format

Expected outputs from your program for each test case are in the answers/ directory. You should print a single line formatted like so:

"hits:16 misses:1 evictions:0"

Indicating that memory accesses led to 16 cache hits, 1 cache miss, and no cache evictions.

---

3. setAssociative: Simulating a 4-way set-associative cache (medium) (24 points)

Your task in this section is to modify your simulator so that it simulates a set-associative cache.

Input format

The input format for the previous two parts, fullyAssociative and directMapped, carry over.

Cache design parameters

For the part of the assignment, your cache simulator should have the following design parameters:

- The cache is an L1 data cache.
- Placement policy: 4-way set-associative cache with 16-byte blocks; with the following parameters:
  - The number of set index bits s=2, such that the number of sets is S=4. Given the choice of s and b, you can think about how to use the memory addresses to deduce which cache set each access must go to.
  - The number of lines per set E=4, making this a 4-way set-associative cache.
  - The number of bits in the block offset b=4, such that the number of addresses in each block B=16.
  - The overall capacity of this cache is therefore still 256 bytes.
- Replacement policy: Least-recently used. If the cache is full, then the cache line that was least recently accessed will be the next victim of eviction.
- Writing policy: write-back and write-allocate.

When the cache simulator starts up, you should assume that the cache is cold, that is, all lines are invalid and nothing is cached.

Output format

Expected outputs from your program for each test case are in the answers/ directory. You should print a single line formatted like so:

"hits:16 misses:1 evictions:0"

Indicating that memory accesses led to 16 cache hits, 1 cache miss, and no cache evictions.

Think about the following:

Notice that all three of the above cache designs are 256-byte caches. Which one had the least cache misses and cache evictions? What was the relative implementation complexity of each design?

---

4. cacheBlocking: Optimizing matrix multiplication using cache blocking (medium) (24 points)

In this fourth part of the assignment, you will improve a program that does matrix multiplication so that it works better with a cache. You will be working with the pa5/matMul/ and pa5/cacheBlocking/ directories for this part. The pa5/matMul/ directory contains a fully written matrix multiplication program pa5/matMul/matMul.c, its pa5/matMul/autograder.py testing script, test cases in pa5/matMul/tests/, and expected answers in pa5/matMul/answers/. The pa5/cacheBlocking/ directory is where you will write your optimized version of matrix multiplication in pa5/cacheBlocking/cacheBlocking.c.

Correctness

First, your matrix multiplication program in cacheBlocking.c should correctly do matrix multiplication. You can use the testing harness in pa5/matMul/ to do this testing. The pa5/cacheBlocking/autograder.py script will also do tests to check for correct matrix multiplication.

Generating memory traces

Second, you can use valgrind to generate memory access traces using this command from the pa5/cacheBlocking/ directory:

```
valgrind --tool=lackey --trace-mem=yes ./cacheBlocking ../matMul/tests/matrix_a_2x2.txt ../matMul/te
sts/matrix_b_2x2.txt
```

Though you can and should just use the pa5/cacheBlocking/autograder.py script, which will call valgrind as above to generate memory traces.

The pa5/cacheBlocking/tests/ directory contains the memory access traces for the baseline pa5/matMul/matMul program that you are competing against.

Simulating memory accesses on a cache simulator

▶, you can use the reference simulator pa5/csim-ref to simulate the memory traces. For this part of the assignment, we assume a 64KB 16-way set-associative LRU cache with 256-byte blocks (a more realistic cache design compared to the example cache designs in the first three parts of this assignment).

The pa5/cacheBlocking/answers/ directory contains the summary statistics for the baseline pa5/matMul/matMul program that you are competing against. You want to optimize your cacheBlocking.c program to perform better than the baseline assuming the above cache design. For full credit, you should have higher hit count than the baseline, lesser or equal miss count than the baseline, and have zero evictions.

Cache blocking

You may want to read about cache blocking, which is the main technique you can use to improve cache performance for this part. Further information is in the class lecture, in the lecture slides, the textbook supplementary slides, the textbook, and in this writeup: **http://csapp.cs.cmu.edu/3e/waside/waside-blocking.pdf (http://csapp.cs.cmu.edu/3e/waside/waside-blocking.pdf)**

---

5. cacheOblivous: Optimizing matrix transpose for better performance with a cache (hard) (24 points)

In this fifth part of the assignment, you will improve a program that does matrix transpose so that it works better with a cache. You will be working with the pa5/matTrans/ and pa5/cacheOblivious/ directories for this part. The pa5/matTrans/ directory contains a fully written matrix transposition program pa5/matTrans/matTrans.c, its pa5/matTrans/autograder.py testing script, test cases in pa5/matTrans/tests/, and expected answers in pa5/matTrans/answers/. The pa5/cacheOblivious/ directory is where you will write your optimized version of matrix transposition in pa5/cacheOblivious/cacheOblivious.c.

Correctness

First, your matrix transposition program in cacheOblivious.c should correctly do matrix transposition. You can use the testing harness in pa5/matTrans/ to do this testing. The pa5/cacheOblivious/autograder.py script will also do tests to check for correct matrix transposition.

Generating memory traces

Second, you can use valgrind to generate memory access traces using this command from the pa5/cacheOblivious/ directory:

```
valgrind --tool=lackey --trace-mem=yes ./cacheOblivious ../matTrans/tests/matrix_a_2x2.txt
```

Though you can and should just use the pa5/cacheOblivious/autograder.py script, which will call valgrind as above to generate memory traces.

The pa5/cacheOblivious/tests/ directory contains the memory access traces for the baseline pa5/matTrans/matTrans program that you are competing against.

Simulating memory accesses on a cache simulator

Third, you can use the reference simulator pa5/csim-ref to simulate the memory traces. For this part of the assignment, we assume a 256-byte 4-way set-associative LRU cache with 16-byte blocks (this design should sound familiar).

The pa5/cacheOblivious/answers/ directory contains the summary statistics for the baseline pa5/matTrans/matTrans program that you are competing against. You want to optimize your cacheOblivious.c program to perform better than the baseline assuming the above cache design. For full credit, you should have higher hit count, lesser miss count, and lesser eviction count than the baseline.

Cache oblivious algorithm implementation

You may want to research about cache oblivious matrix transpose. Cache oblivious code is a way to write programs so that they will have favorable cache performance, regardless of what the specific cache parameters are.

How to submit

From the pa5/ directory, you can run this command to check on the outputs of our autograder script.

```
./assignment_autograder
```

or

```
python3 assignment_autograder.py
```

▶

When you are ready to submit, from the 2021_0s_211/ directory where you see the pa5/ directory, run this command:

```
tar cvf pa5.tar pa5/
```

Upload the file pa5.tar here on Canvas.

By submitting your assignment, you are agreeing to the Rutgers Honor Pledge: “On my honor, I have neither received nor given any unauthorized assistance on this assignment.”

We will not be accepting late assignments. The Canvas submission site will close to enforce the deadline, so be sure to submit early.

If anything is not clear, reach out to your classmates and the instructors on the class Piazza!