CS24 Assignment 5

The files for this assignment are provided in the archive **`cs24hw5.tgz`** on the course website. The archive contains a top-level directory **`cs24hw5`**, in which all other files are contained.

For the questions you need to answer, and the programs you need to write, put them into this **`cs24hw5`** directory. Then, when you have finished the assignment, you can re-archive this directory into a tarball and submit the resulting file:

> *(From the directory containing **`cs24hw5`**; replace **`username`** with your username.)*
> **`tar -czvf cs24hw5-`*`username`*`.tgz cs24hw5`**

Then, submit the resulting file through the course website.

Problem 1 – Cache Analysis *(6 points)*

You are running a program on a 3 GHz processor that has a data cache with 1 cycle of access latency and a main memory with 100 cycles of latency. Your application needs, on average, 1 data value per instruction. Assume the instruction cache is separate from the data cache and the instruction delivery is not limiting your performance.

(a) Your application is completing only 750M instructions per second. Assuming this is primarily due to data memory access effects, what miss rate are you seeing?

Write cache-timing equations and show your work.

(b) If you want to double the performance (instructions per second) of your application, by how much will you need to reduce the cache miss rate?

Put your answers for this problem into a file **`problem1.txt`**, in your **`cs24hw5`** directory.
(Scoring: Each part is 3 points.)

Problem 2 – Cache Simulation *(39 points)*

For this problem you will complete the implementation of a simple cache simulator in the **`cachesim`** directory, and then you will use it to perform some simple cache design analysis.

The cache simulator is a relatively simple program to do basic cache analysis, although the implementation is still rather involved:

- The **`membase.h`** file defines a very simple interface with the **`membase_t`** struct and its associated functions, for a "memory" of some type that can be read and written.
- The **`memory.h`** file provides one implementation of this interface, a simple fixed-size array of bytes that can be read and written. Every read and write is logged in the memory's statistics. The struct that represents the memory is called **`memory_t`**.
- The **`cache.h`** file provides an implementation of a hardware cache as described in class: the cache is comprised of one or more cache sets, each of which contains one or more cache lines. The cache supports writes as well as reads, using a write-back policy for buffering writes, and a write-allocate policy for handling writes to blocks that aren't currently in the cache. The top-

level struct that represents the cache is called **`cache_t`**, and there are several other structs declared in this header file for representing cache sets, cache lines, and the block data within each line.

You will note that the **`memory_t`** and **`cache_t`** structs both start with the same members that **`membase_t`** declares, in the exact same order, but then they also add a few other members afterward. This allows us to cast a **`cache_t*`** or **`memory_t*`** into a **`membase_t*`**, and still see the **`membase_t`** members through the pointer. This is actually the same technique described in the lecture on object-oriented programming; it is a relatively common idiom in C to achieve a generic interface with multiple implementations.

When a **`cache_t`** struct is initialized, its **`read_byte`** and **`write_byte`** function-pointers are set to the cache's version of these functions. When a **`memory_t`** struct is initialized, its function-pointers are set to the memory's version of these functions. Then, we can call these functions via the **`membase_t`** pointer, and it will use the appropriate version based on the function-pointers that were stored in the struct.

Implementation

The cache implementation is not complete. You will need to complete the implementation of several important functions before you can use it in the simulator. All functions to be implemented are in the file **`cache.c`**. You can look at the function **`init_cache()`** to see how caches are set up.

(a) The **`decompose_address()`** function takes the address of a memory access, and determines the cache-set, the tag, and the offset within the block, that the address corresponds to. This function is used in both read and write operations. You will need C shift and bitwise logical operations to implement this function. *(5 pts)*

Note that both the block size and number of cache-sets are powers of 2. This means that it's very easy to construct bit-masks to mask out different parts of the incoming address. For example, if the block size is 32_{10}, or 00100000_2, the block offset will occupy 5 bits of the address. We can easily construct a mask by computing $32_{10} - 1 = 00100000_2 - 1 = 00011111_2$, which has the five bottommost bits set.

(b) There are three functions that determine block addresses and offsets based on various details. Complete the implementations of all three. *(3 pts per function, 9 pts total)*

The **`get_block_start_from_address()`** function takes a memory address and determines the start of the block that the address falls within. This is used in cases where a memory access results in a cache miss, and the block containing the address must be loaded into memory.

The **`get_offset_in_block()`** function takes a memory address and determines the relative offset of the access, with respect to the start of the block. This is used to handle both reads and writes to a given cache line, once it contains the block data corresponding to the address.

The **`get_block_start_from_line_info()`** function takes the tag from a cache line, and the index of the cache-set the line was in, and determines the start of the block stored in the cache line. This is used in cases where a dirty cache-line must be evicted, and it must be written back to the next level of the cache.

(c) The **`find_line_in_set()`** function is used after a cache set is identified, to see if a line has the specified tag. Recall that the valid flag must also be set for the line to be in the cache. If no line contains the specified block, the function returns **`NULL`**. *(6 pts)*

Caching hardware implements this operation as a fast associative memory, but you should implement this function by looping over the cache-set's contents.

Once you have completed these steps, you should be able to build and run the **`testmem`** program to verify that the cache implementation is working properly. The program sets up a memory with one level of caching, performs many writes to the cache, then flushes the cache and verifies that the memory has the expected state after performing all writes.

Measurements and Replacement Policy

Three different algorithms are implemented against the cache simulator:

- **`qsorttest`** implements an in-place quicksort of a large array of integers.
- **`heaptest`** implements a heap data structure (search for "binary min heap" on the Internet for details) which is used to sort a large array of single-precision floating-point numbers.
- **`apsptest`** implements the Floyd-Warshall All-Points-Shortest-Path algorithm, for finding the shortest distance between any pair of nodes within a graph. This implementation includes a *path-reconstruction* modification that allows the actual shortest path between any pair of nodes to be reconstructed. This algorithm is very slow ($O(N^3)$ for N vertices); when you run it, it prints out one dot for each N^2 pass over the graph's weights.

Each of these programs creates a memory of the required size and then runs the algorithm against the memory. At the end of the run, the program prints out various memory usage statistics.

To run any of these tests with a cache, one or more command-line arguments of the form B:S:E can be specified, where B is the block size in bytes, S is the total number of cache-sets, and E is the number of lines per cache set. Both B and S must be powers of 2. For example, to run the heapsort-test program with a cache of 64-byte blocks, 1024 cache-sets, and 4 cache-lines per set (a total cache size of 256KiB), this command-line could be used: **`./heaptest 64:1024:4`**

Note that these programs are also capable of generating different random data at each run, or using the same data on each run. The following measurements should be performed with the **`SEED`** value set to 54321098, which should be how the files are when you receive them. (You only need to verify this if you decide to tinker with **`SEED`** for the sake of your own curiosity.)

(Put your measurements into a file **`cachesim/problem2.txt`**.)

(d) Run the **`heaptest`** program with a direct-mapped cache of size 8192 bytes, and a block size of 32 bytes. Record the command-line invocation for the program execution, including your cache parameters. Also, record the cache miss-rate reported by the program. *(3 pts)*

(e) Again, run the **`heaptest`** program with a fully associative cache of size 8192 bytes, and a block size of 32 bytes. Record the same details as for the previous problem. (Note: This run will be significantly slower, since our cache simulator has a terribly inefficient mechanism for finding the cache-line with a particular tag.) *(3 pts)*

You might notice that the results in part (e) are really no better than the results in part (d), even though a fully associative cache should be more effective than a direct-mapped cache! This is

because the cache currently uses a random replacement policy; when a line must be evicted from a cache-set, the cache simply picks one at random. This is inefficient; it pays no attention to how the program is accessing memory.

(f) Modify the cache implementation to use a Least Recently Used replacement policy. Do this by adding a new field to the cache-line struct, which stores the most recent time that each line was accessed. (This value should be of type **`unsigned long long int`**, which will be 64 bits.) *(10 pts)*

In the cache's **`cache_read_byte()`** and **`cache_write_byte()`** functions, update this "most recent access time" value on the cache line being accessed, every time a byte is read or written on the cache line. You can use the **`clock_tick()`** function declared in **`membase.h`**; every time this function is called, it increments a counter and returns the new value. Thus, it simulates a simple clock.

Finally, update the **`choose_victim()`** function to implement the LRU policy. Scan through the cache lines:

- If you come across a line that is invalid, just return that line immediately because it's currently empty.
- Otherwise, find the line with the smallest access-time value; this is the line used furthest in the past, and therefore is the "least recently used."

Once you have completed this work, set the **`RANDOM_REPLACEMENT_POLICY`** constant at the top of **`cache.c`** to 0 to switch to LRU. Perform a clean rebuild of your code, and try the **`testmem`** program again to verify your implementation.

(g) Run your tests from steps (d) and (e) again, including your command-line invocations and the results. You should see a modest improvement in the fully associative cache test, due to the cache making better decisions about what lines should be evicted. *(3 pts)*

(Again, put your measurements into a file **`cachesim/problem2.txt`**.)

Problem 3 – Cache Usage Optimization *(55 points)*

*Files for this problem are in the **`multimap`** directory.*

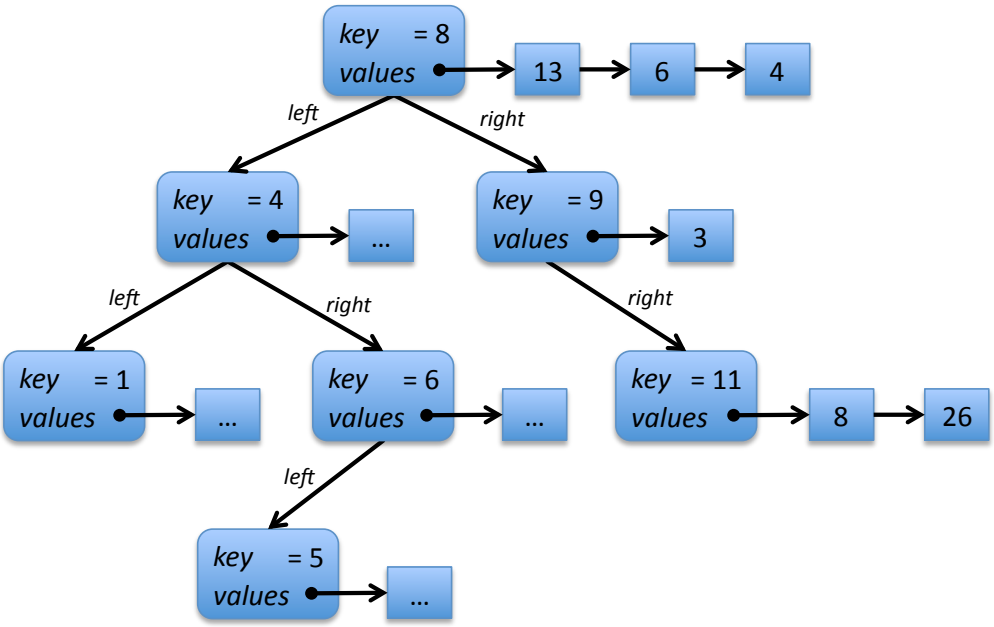
A ***map*** is a data structure that stores (*key*, *value*) mappings, but each key may only appear once in the map. A ***multimap*** relaxes this constraint and allows a given key to appear multiple times. There are several ways to implement a multimap; one is to use a hash table to store (*key*, *value*) pairs, and the other is to use some kind of tree structure. The second option maintains the (*key*, *value*) pairs in order by key, which allows the pairs to easily be traversed in increasing order of key.

For this problem we will explore a simple ***ordered multimap*** implementation that uses a binary tree structure. Every unique key-value is represented by its own binary tree node. As is typical for binary trees, the left child of a tree node only holds keys that are strictly less than the node's key, and the right child of a tree node only holds keys that are strictly greater than the node's key.

The values associated with a given key are stored in a linked list that the key's corresponding tree-node maintains. The list may contain multiple instances of a given value, and the values can be in

any order. (We only care about traversing the multimap in order based on keys.) As (*key*, *value*) pairs are added to the multimap, a given node's list may have many values in its linked list; as (*key*, *value*) pairs are removed from the multimap, a node's list of values may become empty; at this point, the node is removed from the binary tree.

Here is a diagram that illustrates the basic concepts behind the data structure:



Each tree node holds a pointer to its left and right child. Each node also holds a specific key-value, and a linked list of values associated with the key. Keys and values are both 32-bit integers.

An implementation of the above data structure is provided in **`multimap.h`** and **`mm_impl.c`**. Two test programs are provided; **`mmtest.c`** performs some basic correctness tests, and **`mmperf.c`** performs various performance tests.

(a) In the directory **`cs24hw5/cpuinfo`** is the source for a program **`cpuinfo`** that reports the processors and caches of the machine it is run on. Run this program on the machine you will be using for performance testing, and save the output into a file **`cs24hw5/multimap/cpuinfo.txt`**, e.g. by running:

*(from the **`cs24hw5/cpuinfo`** directory, after building **`cpuinfo`**)*
```
./cpuinfo > ../multimap/cpuinfo.txt
```

(2 points)

(b) Build and run the tests. The performance tests only focus on read performance, which means that we are specifically measuring the memory-access performance of the data structure, not allocation performance. (Normally we would also optimize insert/remove performance, but then our measurements won't capture only cache performance.)

Note that **`mmperf`** measures the *wall-clock time* of your program running (i.e. how much time actually passes), not the actual number of CPU clocks or instructions executed or anything like

that. This means that the performance measurements may be affected by other programs running on your computer at the same time, as well as more subtle things like whether the terminal window currently has the focus. When you are generating your performance measurements, close down any other compute-intensive programs or media-player programs that may be running, then run your program and don't do anything else while it runs.

In the file **`questions.txt`**, paste the output of running **`mmperf`** on your computer.

(2 points)

(c) In this same file, explain what part of the multimap is being exercised by the first three tests, and what part of the multimap is being exercised by the second three tests. (Hint: You will notice that in each set of three tests, keys are added randomly, in increasing order, and in decreasing order. This is because our basic multimap implementation does not ensure that the tree is kept balanced, so tree structure and depth is affected by the order that keys are added.)

(10 points)

(d) The performance of this data structure can be *greatly* improved, simply by making it more cache friendly. Copy **`mm_impl.c`** to **`opt_mm_impl.c`**. Then, modify the implementation in **`opt_mm_impl.c`** to be much more cache friendly. You can generate test programs for your optimized multimap with the command "**`make opt`**". This will generate **`ommtest`** and **`ommperf`**, which will use your optimized multimap implementation instead of the default implementation.

Depending on your system, you should be able to achieve at least these performance improvements:

- Tests 1-3 should see anywhere from 20x to 50x performance improvement.
- Test 4 should see at least ~10x performance improvement.
- Tests 5-6 should see anywhere from 20x to 50x performance improvement.

Full credit will be awarded for achieving the upper end of these ranges; worse results will receive a corresponding deduction. Clever, highly optimized submissions will likely result in bonus points. Of course, **`ommtest`** should also report zero failures for your implementation!!!

You may change anything you want about the data structure, as long as you keep the following constraint in mind: **Your efforts should be focused on improving the cache-friendliness of the data structure.**

- For example, you cannot change the multimap to use hashing; the keys should continue to be kept in sorted order. (If you wish to use a different representation of the tree structure, this is fine. If you wish to maintain a balanced tree, that is also fine.)
- Values should continue to be kept in no particular order. (If you wish to reorder values, e.g. during value-removal, this is fine, but do not change the lookup mechanism; it should linearly search for the value in the sequence of values associated with the key.)

If, during implementation, the tests run too slowly for your machine, you can alter the values at the top of **`mmperf.c`** to make them faster. **Make sure to return these values to their defaults before running your performance tests to turn in.**

- Setting **`EXCLUDE_SLOW_TESTS`** to 1 will disable the last two tests, which are significantly slower on some machines.
- Decreasing the **`SCALE`** value will run proportionally fewer probes in each test. Correspondingly, increasing **`SCALE`** will run more probes in each test, in case your tests run too fast for your preference.

Finally, the **`Makefile`** has properties for building debug and non-debug versions of the multimap; after you have finished implementing and debugging your code, turn off debugging for your final measurements.

(25 points)

(e) Again in the file **`questions.txt`**, explain all optimizations you implemented in your version of the multimap. For each optimization, include the following:

- What cache-performance issue in the original implementation you are trying to address.
- A description of your optimization.
- An explanation as to why your optimization mitigates the problem you identify.

(14 points)

(f) Finally, include the result of running **`ommperf`** in **`questions.txt`**.

If you altered any of the **`#define`**s in the **`mmperf.c`** code, restore them to their defaults before running your performance test. Additionally, double-check your **`Makefile`** to ensure that you aren't performing a debug build; this will enable compiler optimizations and disable some debug-mode code.

If any of these things needed tweaking, make sure to perform a clean rebuild before your test!

(2 points)