Jillian James
Marika Swanberg
CSCI 389

ReadMe for HW5

Part 0: (Running the files)

To run the files, first compile the benchmark and client files using:

```
g++ -o benchmark benchmark.cc client.cc
```

Then compile the files on the server side:

```
g++ -o server server.cc cache.cc
```

To run first initialize the server like so (For this homework, -m maxmem may be varied at values over 1000 but -t portnum must be 8080 for the client and server to communicate. Not included these parameters sets maxmem to 100 and portnum to 8080.):

```
./server -m 10000 -t 8080
```

Open another terminal window to run the benchmark on the server:

```
./benchmark <number of items in cache> <requests per second>
```

Part 1:

In this homework, we evaluate the cache and network system that we have created for CSCI 389 over the past few weeks. Our system involves a client and a server. On the client side, the client initialize a cache that uses http requests On the server side the server initializes a cache using the cache.cc file developed throughout HW2 and HW3. When the client makes a request (a get, a set, or a delete) this request is sent to the server via http requests, and the server executes these requests on its cache implementation. The goal of this assignment is to evaluate sustained throughput and the latency of get requests as a function of how many items are in the cache. The workload we use is a combination of get,set,and delete described in part 7. This performance evaluation was run on a machine using:

OS: Ubuntu 17.10
Memory: 3.9 GiB
Processor: Intel Core i5-7267U CPU @3.10GHz
OS Type: 64-bit
Disk 66.3 GB

Part 2: Services and Outcomes

The services for our system are cache operations defined in the API. These include: get, set (update or insert new k-v pair), delete, spaceused.

- Get will either return a pointer to the associated value and set the size parameter passed by reference to the size of the value it's pointing to, or if the entry is not in the cache, the function returns a nullptr and sets the size to 0.
- Set, when given a key and reference to a value, will check whether the key is already in the cache and either insert the new entry or update the old one accordingly. It returns 0 if there were no errors and 1 if an error occurred.

- Del (delete), when given a key, will delete the associated entry in the cache and free the memory and returns 0. If the key is not in the cache, del does nothing to the cache and returns 0.
- Spaceused returns the total number of bytes of memory that the values in the cache take up. Our cache implementation uses an ordered map, but the value spaceused returns does not take the ordered map's data members of size of its keys into account.

Part 3: Metrics
One of our metrics will be *sustained throughput*, defined as the maximum offered load (in requests per second) at which the mean response time remains under 1 millisecond. We will also measure the latency of get requests as a function of how many items are in the cache.

Part 4: Parameters

System parameters: see hardware description above
Workload parameters: maximum cache size (maxmem), set:get ratio, requests per second, the sizes of items in the cache

Part 5: Factors and levels
For the sustained throughput experiment we varied the requests per second. These levels were: 780, 1317, 2333, 4604, 7028, 11264, 24828, 65710, 65178, 59449, 68031. The values we ended up graphing were the real requests-per-second the program was actually able to carry out. We explain why we were not able to accomplish the exact requests per second we input in part 9.

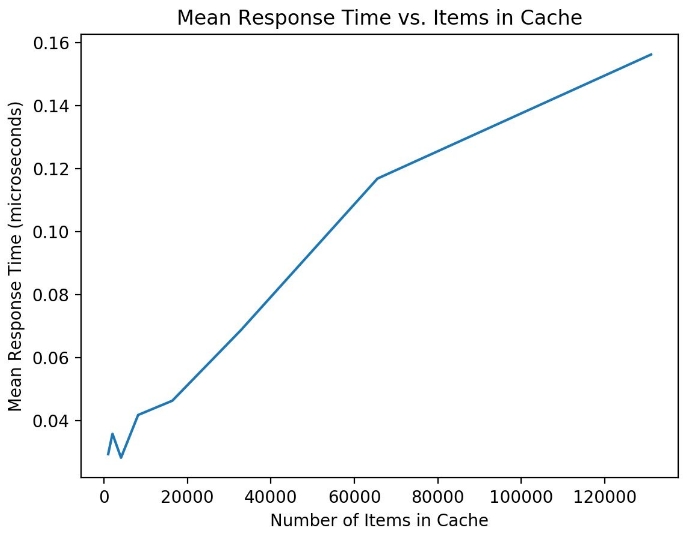
For the average response time as a function of the number of items in the cache, we varied the number of items in the cache by powers of two from 2^10 to 2^17.

Part 6: Technique
We will be doing a simulation of a cache workload on our cache. It is a simulation because we will be generating a synthetic workload consisting of a combination of get, set and delete to test the cache rather than testing it on real-user usage of the cache.

Part 7: Select workload
The workload we used is a combination of get, set, and delete. 96% of the workload is gets, 3% of the workload is sets, and 1% of the workload is delete operations. We accomplished this in benchmark.cc using randomly generated numbers which we stored in two vectors. The first vector determines which operation will be performed, and the second vector determines which item to get if that is the chosen operation. We chose this particular workload because we wanted to mimic the 30:1 get to set ratio used to test memcache, and we used randomness so that the workload is more realistic. We chose to leave spaceused out of our workload because we thought it was a trivial operation.

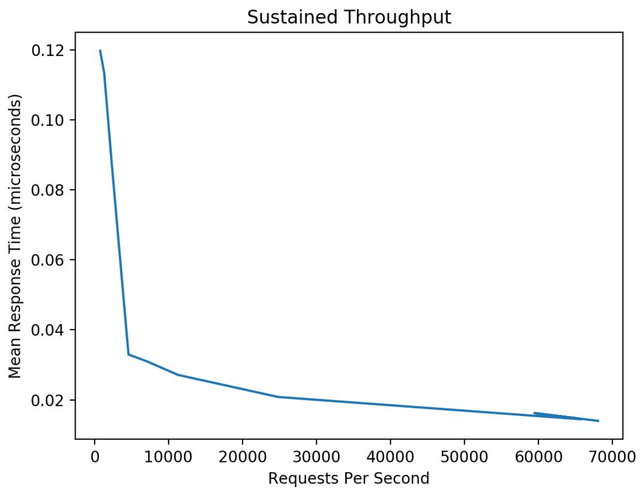
Part 8: Experiment design
Before timing our cache operations, we fill up the cache with key-value pairs like “0”:”0” …. “n”:”n” where n is the number of items we want in the cache. In our benchmark.cc we run the benchmark ten times. Within the benchmark we have a loop that is supposed to execute as many get, set, or delete operations as it can in the ten second period. To accomplish this, we use the target requests per second to determine how many microseconds each request should take. We then have the program sleep if it can complete the operation faster than the allotted number of microseconds. We believe this should make it so that each run of the program executes in 10 seconds, but we found that in practice it takes longer than that. We speculate on why this happens in Part 9.

Part 9: Analyze and interpret results
We had some problems with our benchmark, and did not time all of the points we had originally set out to time. We discovered that as our requests-per-second or the number of items in the cache increases, that the time it takes to run the benchmark increases even though we were not achieving the desired requests-per-second. We calculate the actual number of requests-per-second that our code achieves and use that to plot our results, rather than the value we inputted because the code (and thus, the number of requests per second) was always slower than we wanted. We believe this is because there is some time factor we did not account for, so our program waits longer than it should and thus cannot reach the desired requests-per-second. We found an upward trend in our throughput experiment but could not find the maximum number of requests per second at which response time remained under 1 millisecond. We could not find this value because our tests began to take much too long at higher values due to the mysterious timing issue described above, and were unable to achieve the throughput we wanted. Below we show a graph for each of our experiments:



This figure shows a general linear upward trend in the response time for each operation as the number of items in the cache increases. Note that the items in the cache are all of equal size, because they are all strings.

This trend matches the expectation, since any get operation is going to take time linear in the size of the cache, and our workload is mostly composed of gets.



This figure shows a downward trend in the mean response time as the number of requests per seconds increases (for a starting value of 10,000 items in the cache). We think that this trend occurred due to decreased overhead as the number of requests and responses increases.

Part 10: Present results

We were unable to find the throughput threshold but found the general upward trend pictured in the first graph, and discovered the issues explained in Part 9. Given more time, we believe we would be able to resolve the timing issue in Part 9 and find the throughput threshold.