Multiprocessor Synchronization Capstone Writeup

Tristin Falk-LeFay, Hans Sun

1. Introduction

We will explore the impact of different concurrent hashmap implementations on the performance of a parallel firewall application. The firewall is responsible for filtering incoming packets, and possibly discarding sketchy packets.

In particular, the firewall is responsible consuming packets of two flavors: data and configuration. Configuration packets update the internal permissions of the firewall, while data packets are filtered according to the internal permissions of the firewall, and, if approved, require some amount of work to be processed entirely. The internal permissions of the firewall consist of two HashMaps. The first, PNG, maps packet addresses to booleans; if an address maps to True, this indicates that the address has permission to send packets. Otherwise, it cannot. The second, R, maps addresses to a set of other addresses; the set of addresses that a particular address maps to indicates which addresses that address has permission to receive from.

We have two firewall implementations: serial and parallel. As one may guess, the serial firewall handles packets one at a time, and the parallel firewall handles packets in a multithreaded fashion. This is described in more detail in the 'Design' portion.

For the parallel firewall, we chose to experiment with 5 different implementations of concurrent hashmaps with which to maintain permissions:

1. Java built-in ConcurrentHashMap.
2. Github Open Source NonBlockingHashMap
3. Coarse-grained HashMap, derived from the textbook's Coarse-grained HashSet
4. Striped HashMap, derived from the textbook's Striped HashSet
5. Coarse-grained Java Map, which is simply a non-concurrent hash map wrapped with Synchronized methods.

2. Design

For the serial firewall, we have one public method: handlePacket(Packet pkt) which in turn calls one of two private methods depending on the type of packet: handeConfigPacket(Packet pkt), handleDataPacket(Packet pkt). The first updates permissions, the second checks permissions and performs work. The serial firewall is fed one packet at a time.

For the parallel firewall, we have the same public method: handlePacket(Packet pkt). The primary difference is that this method offloads the processing of the packet to one of two thread pools: configPool or dataPool. Each pool is responsible for processing it's respective type of packet. We wrapped the handleConfigPacket and handleDataPacket functions from the sequential firewall in Runnable classes, making the distribution of work very simple: we just create the Task within handlePacket, and submit it to the respective thread pool.

In order to make testing as easy as possible, we pass in a few parameters to the ParallelFirewall when instantiating it. These include which type of map it should be using for the permissions maps, as well as the number of threads to use. By default, the firewall splits the number of threads available equally between the configPool and dataPool, with an extra thread going to the dataPool in the case of an uneven number of threads.

3. Testing

There were multiple parameters that we changed while testing the parallel firewall versus the serial firewall. Primarily, these were:

1. The PacketGenerator configuration.
2. The number of threads being used. (we used a 16-thread Google Cloud Instance)
3. The type of map to use.

For each combination of the 3 above we ran several trials and recorded how long it took to process 10,000,000 packets. We calculated speedup as the speed of packets processed (packets/ms) of the parallel firewall divided by the speed of the serial firewall. A throughput of 1 means it is on par with the serial firewall, with > 1 being better and < 1 being worse.

The PacketGenerator is the source of the packets. The PacketGenerator has many parameters, but we chose to base them off of the 8 described in the handout.

We've created a graph of the throughput of each map versus the number of threads for each PacketGenerator configuration below.

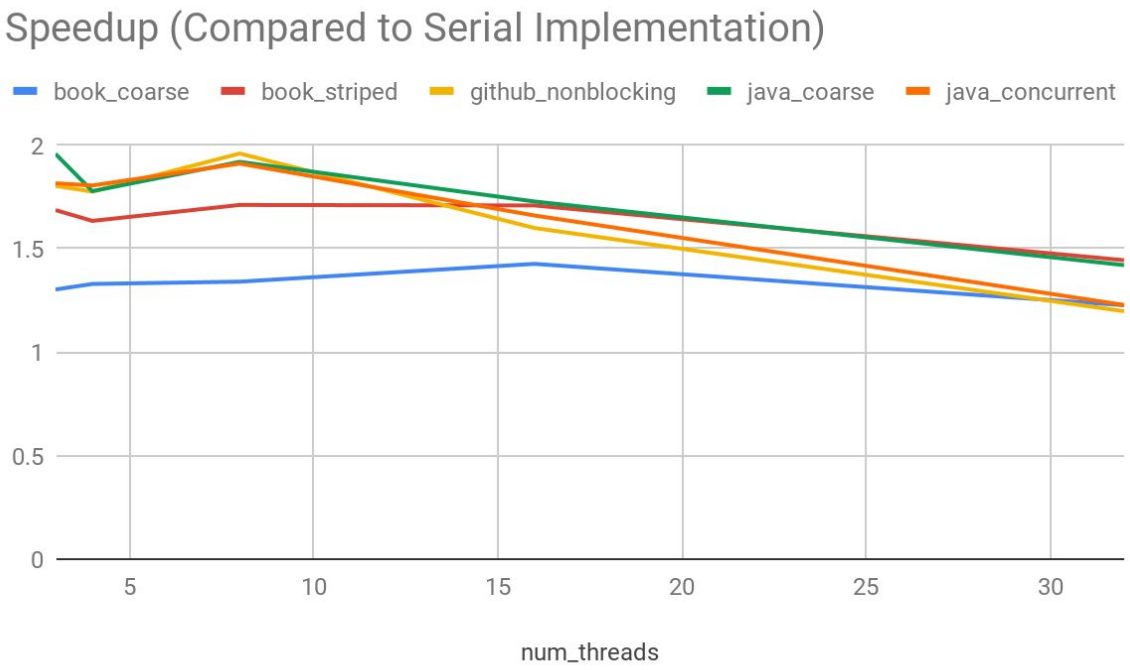
4. Results + Conclusion

For the most part, the parallel firewall outperformed the serial firewall. Depending on the parameters of the packet generator, number of threads, and map type, the speedup ranged from slightly below 1x, up to 6x. As expected, most maps had the best performance when running with 16 threads, since we were running our experiments on a 16-core machine. Generally either the java built-in ConcurrentHashMap or the nonblocking map from Github performed the best, while the two implementations from the books performed the worst.

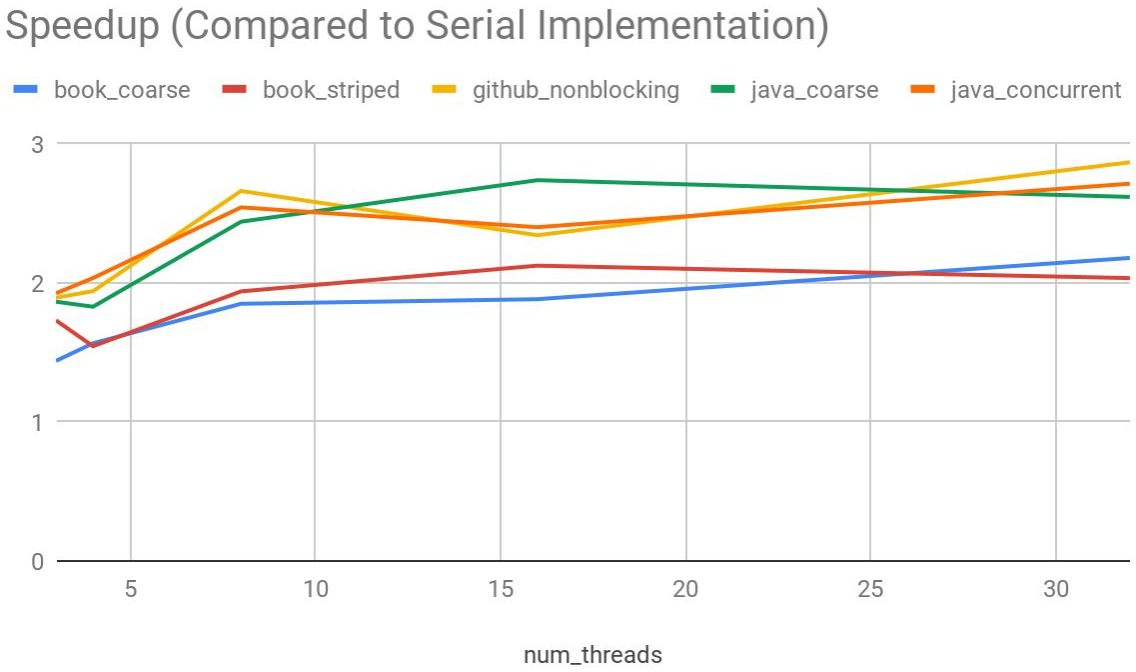
While the parallel firewall fared better in general, how much better largely depended upon the packet generator. We hypothesize that this is due to the fact that as the packet generator parameter mix number increases, the balance between the number of config packets and data packets approaches 50/50. Since our parallel firewall was built to split threads equally between both of the pools, it makes sense that it would perform better on a balanced generator, as that would maximize utility of worker threads.

5. Visualizations

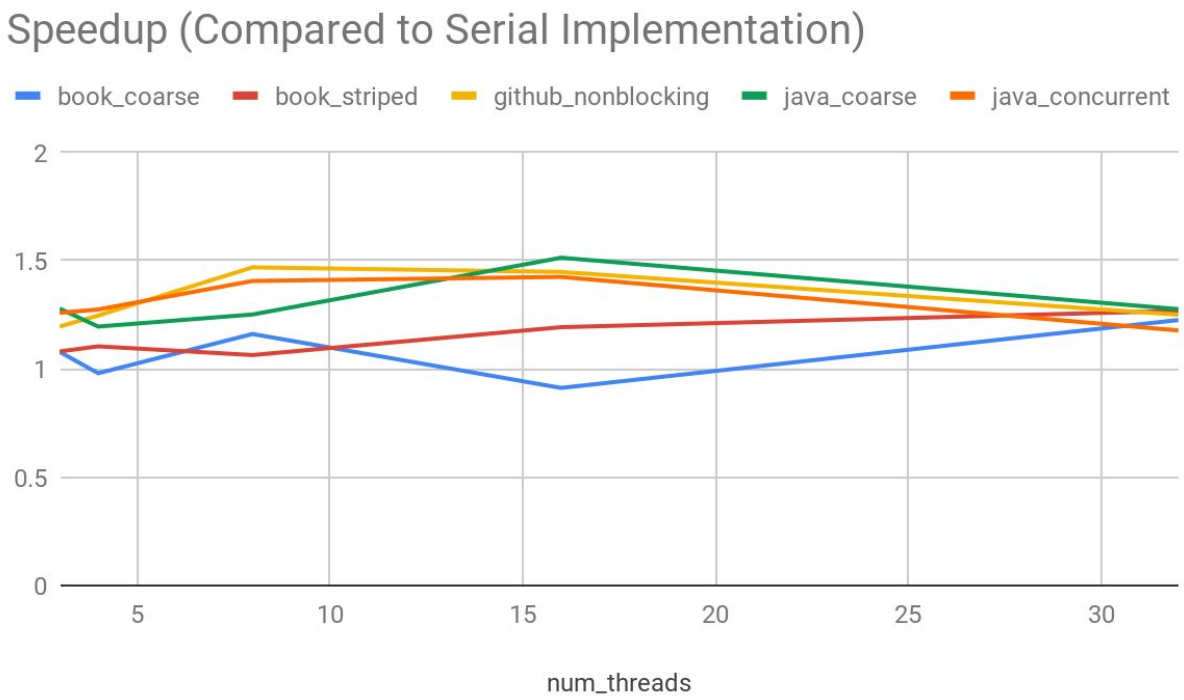
Parameter mix 1:



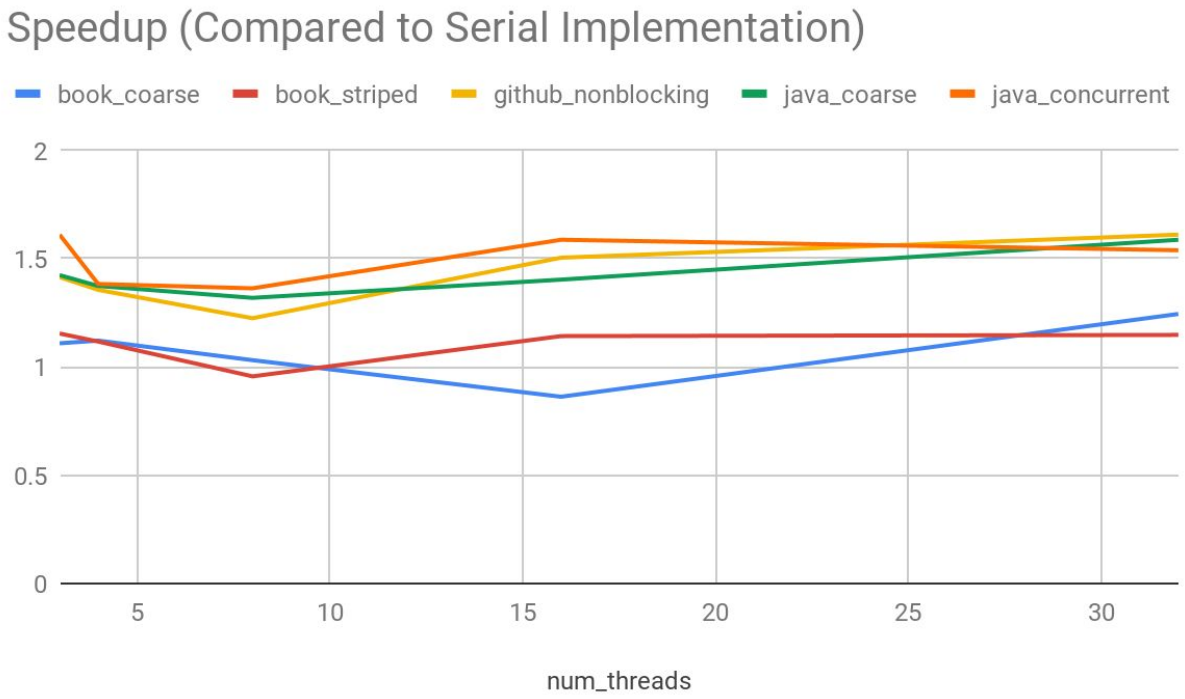
Parameter mix 2:



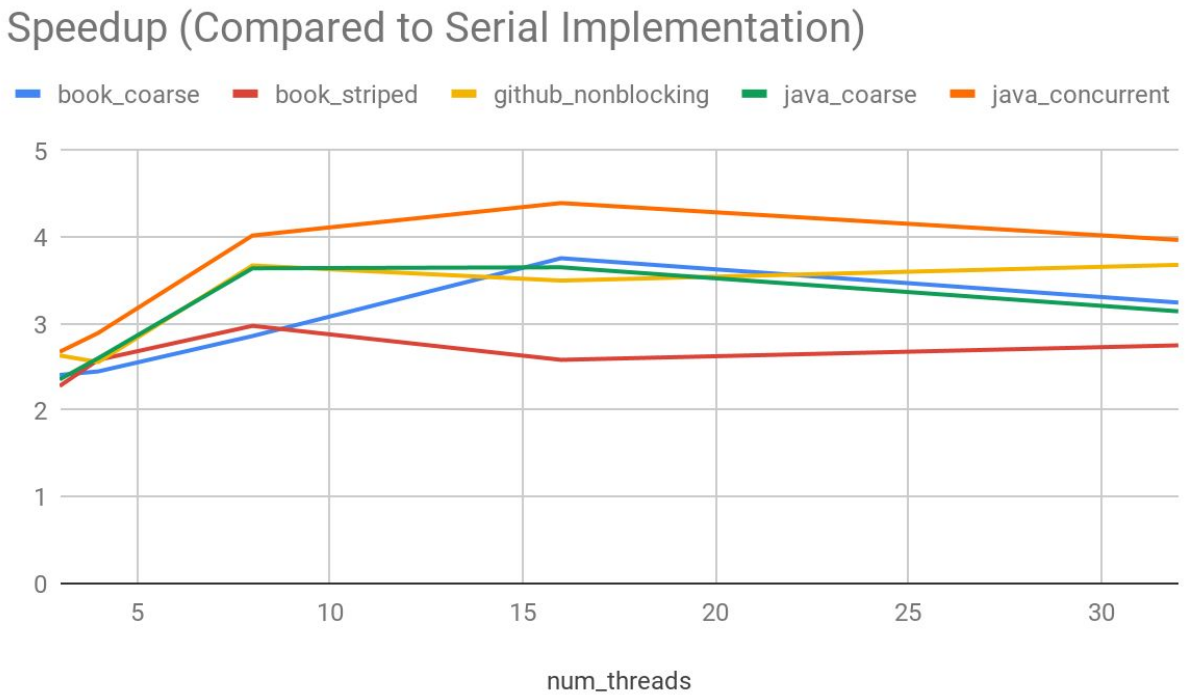
Parameter mix 3:



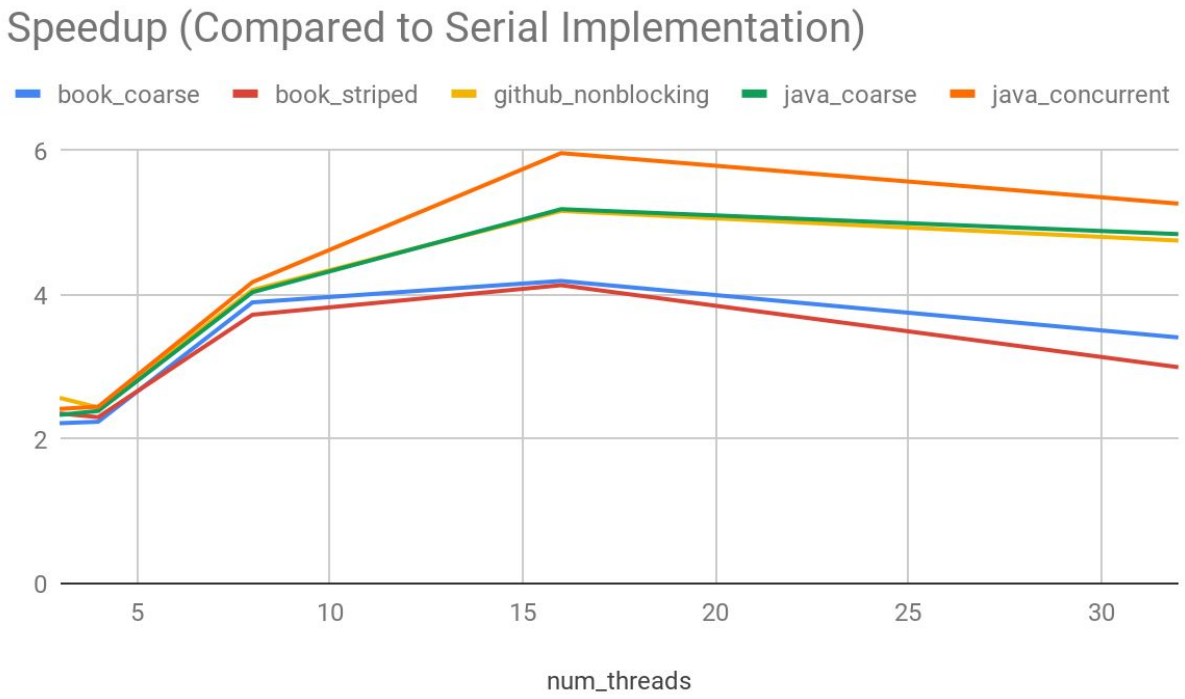
Parameter mix 4:



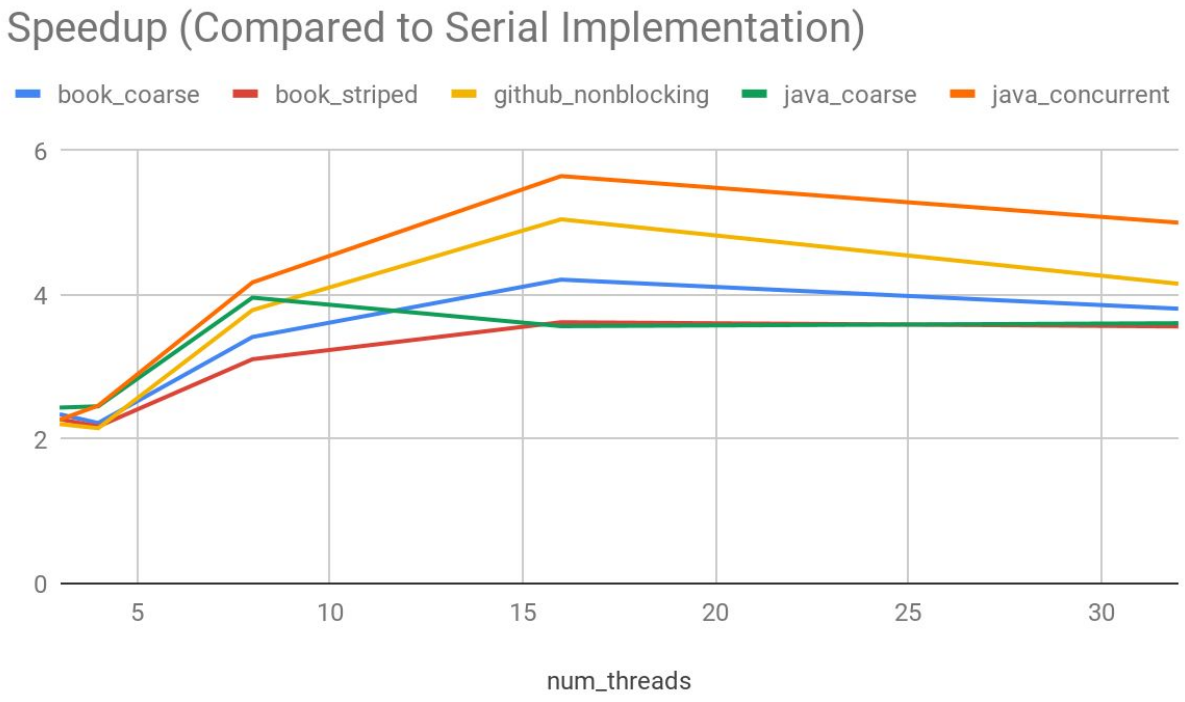
Parameter mix 5:



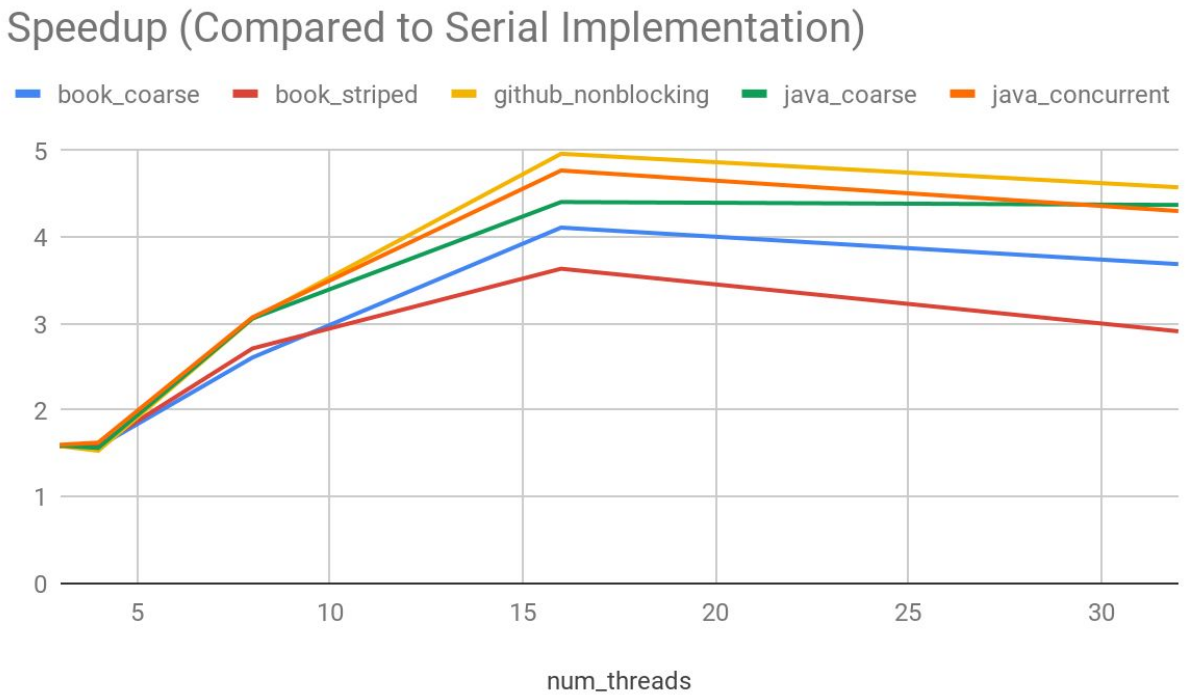
Parameter mix 6:



Parameter mix 7:



Parameter mix 8:


Speedup (Compared to Serial Implementation)
book_coarse
book_striped
github_nonblocking
java_coarse
java_concurrent
5
4
3
2
1
0
5
10
15
20
25
30
num_threads

6. Advice for Future Students

There were a few things that needed to be done in order to make the code given in the jars completely functional. We ended up using the following files:

- Fingerprint.java
- Firewall.java
- PacketGenerator.java
- RandomGenerator.java
- StopWatch.java

as well as any files that the above depended on. We ended up having to modify PacketGenerator because the addresses that were being generated for the packets had different ranges for the config and data packets, so that the config packets were never modifying relevant addresses. We fixed this by removing the call to mangle() in one of the methods so that the addresses generated for each type of packet were in the same range since the other method did not have the mangle() call. We created all other necessary java classes that we could not find code for.

In terms of actually implementing the different hashmaps used to distribute work and keep track of addresses, we found the book useful for the coarse-grained and striped hashmaps. We also found and used different built in implementations as well as an implementation for a lock-free hashmap someone had made on GitHub.

Since we were testing our program on differing numbers of threads, we used a 16 CPU Google Compute instance to make sure that we had accurate data for our program. We used GitHub for version control and pulled to the instance when we had to gather data.