Capstone Project Final Report

Multiprocessor Synchronization - Prof. Maurice Herlihy

Iuliu Balibanu & Samantha Mayfield

December 2017

Contents

1 Project Overview

For our capstone project, we implemented several variations of a firewall which receives packets of information from a range of addresses and processes them in parallel. We used a mix of custom and builtin Java data structures to test the performance of several implementations.

1.1 Packets

Packets are provided by the PacketGenerator. There are two types: Data packets and Config packets. Data Packets store their data and their source address and destination address. Config packets contain information to update the permissions of addresses and change which addresses are allowed to communicate.

1.2 Permissions

There are two types of permissions:

1. **Persona-non-grata**: Some addresses are not allowed to send packets to any other addresses. These are considered persona-non-grata addresses, and no data from them should be processed.

2. **Accepting Range**: Some destination addresses have an Accepting Range and are only allowed to receive information from certain source addresses.

1.3 Concurrency

Given the volume of packets, one of the most important aspects of this project was to process packets in parallel. The baseline PacketGenerator itself cannot be accessed concurrently, so we created a threadsafe (course-grained locked) version to feed our threads and to produce a parallelizable stream.

2 Implementation Details

We focused on two major aspects of a Firewall's functionality:

1. How the Firewall takes a serial stream of packets from PacketGenerator and sends these packets to threads for processing. This was mostly a problem of multithreading style and parameter tuning.

2. How the permissions for each source and destination pair are maintained and checked. This was an intensive exercise in data structure implementation and benchmarking.

As our baseline, we used a naive serial firewall built on top of basic (serial) Java data structures.

2.1 Packet Processing

We implemented packet processing using both a traditional thread-based approach, and using Java's Parallel Streams.[1]

1. Threading: The performance testing was carried out on an Intel I7 processor with 8 logical cores, so we used 8 threads which all retrieved packets from a thread-safe packet generator, and incremented an atomic counter after each packet was processed until the counter reached the target number of total packets. We then joined the threads and halted execution.

2. Streaming: Using the same thread-safe packet generator as above, we created a Stream of packets which we then processed in parallel (using *Stream.parallel*), limiting the length of the stream to the total number of packets we wished to process.

   In comparison to the threaded implementation, the streamed implementation was notable for its clean syntax and lack of *for* loops.

2.2 Permissions Checking

To maintain the addresses from which each address will accept packets, we alternatively used three different data structures:

2.2.1 The Builtin ConcurrentHashMap

We used the builtin Java ConcurrentHashMap as a baseline for concurrent performance. It maps destination addresses to a list of Range objects. Each Range represents a range of source addresses from which the given destination address will not accept packets (by default, all addresses are accepted).

The ConcurrentHashMap was a good solution in that it was simple to plug in and has been optimized for us by much better concurrent programmers. However, we were limited by the functionality it provides. For example, we would have liked to use an atomic testAndSet method that can add an item to the HashMap if it wasn't already present, but the ConcurrentHashMap doesn't expose this functionality. Additionally, while getting the list of Ranges for a given destination address is a constant time operation, finding a source address in that list might take much longer if there are many sources that the destination will not accept.

2.2.2 A Custom Striped HashMap

We created our own HashMap implementation using striped locking [2]. We chose this because it allows us to easily resize the table and offers far less contention than a coarse-grained locked Map. In practice, very few threads simultaneously access the same bucket, so we felt a lock-free implementation would not be worth the necessary overhead. Like the ConcurrentHashMap, we store destination addresses as keys and lists of Ranges as values. The HashMap is implemented as an array of lists, with each list containing our own HashablePair objects, which each contain a key and a value. We store these instead of the values themselves to easily recover the key that was associated with a value after the key has been hashed. To insert an item into the HashMap, we assign an address to a bucket by taking the hashCode() of the address modulo the length of the array.

To ensure accurate concurrent access, the HashMap keeps a second array of ReentrantReadWriteLocks. Each lock is associated with a bucket, so a thread calling get() on the HashMap must acquire the read lock for the bucket mapped to by its hash, and likewise a thread calling put() must acquire the write lock for the corresponding bucket.

To resize the HashMap, a thread first acquires all of the write locks by iterating across the array. Then, it doubles the array size, re-hashes the elements in the old array, and releases all of the locks. The HashMap takes advantage of lock striping to avoid ever needing to resize the lock table. After a resize has occurred, each lock corresponds to twice as many buckets as it did before the resize.

For our purposes, the striped HashMap was beneficial because we could fine-tune its parameters to our requirements. For example, we can control the initial size, maximum length of an individual bucket, and threshold to resize. We cannot control these parameters for the ConcurrentHashMap; we are stuck with its default settings. Additionally, we were able to implement an atomic testAndSet method called replace(), which we would have liked to have for the ConcurrentHashMap. However, as with the ConcurrentHashMap, we are limited by the runtime of searching through a list of ranges for a specific source address.

2.2.3 A Custom 2D Boolean Array

Our final solution to maintaining a list of permissions is a 2D array of booleans. Each source address has a row, each destination address a column, and their intersection stores whether that source is allowed to communicate with that destination.

Each column has a ReentrantReadWriteLock. To read or update the permissions for a particular destination address, a thread must acquire the corresponding lock.

One advantage to the 2D array is that it doubles as the data structure which contains persona-non-grata information, which saves the need for a separate structure. This implementation proved to be faster than either of the HashMap implementations, as finding the permissions for a source destination pair can be done via a single array lookup. Additionally, the locking mechanism provides slightly lower contention than the StripedHashMap, where each bucket might contain information for several addresses, since in the 2D array we have one lock per destination address. However, this implementation necessitates a space-time tradeoff, as the number of booleans in the array is equal to the square of the total number of addresses. Furthermore, it is limited by the fact that its use depends on knowing ahead of time your possible address range, making it unusable for unbounded addresses, and very inefficient if you *could* have a wide range of addresses but in reality usually only have a small number of unique addresses during any given firewall execution.

3 Performance Results

We tested the performance of our Firewall with all six combinations of inputs: each of our accepting range permissions implementations (the builtin ConcurrentHashMap, our StripedHashMap, and the 2D array) with both methods of packet processing (threads and streams). The below figures show some of our results from these trials.

3.1 Methodology

All of the figures presented in this section show average runtime vs. number of packets processed, unless otherwise noted. We break out our results into *critical runtime*, which is defined as the time taken to process threads after initializing relevant data structures,

and *total runtime*, time from start to finish. Average runtime is calculated by looping the Firewall through the given number of threads 100 times and taking the average of all results. This is done twice for each value and the second of these results is what is plotted on the charts shown. See the data appendix for the full data tables used in the construction of these charts.

3.2 Serial Implementation vs Parallel Implementations

Overall, and as expected, all of our parallel implementations were significantly faster than the serial implementation. Our parallel implementations generally ran six times faster than the baseline serial implementation.

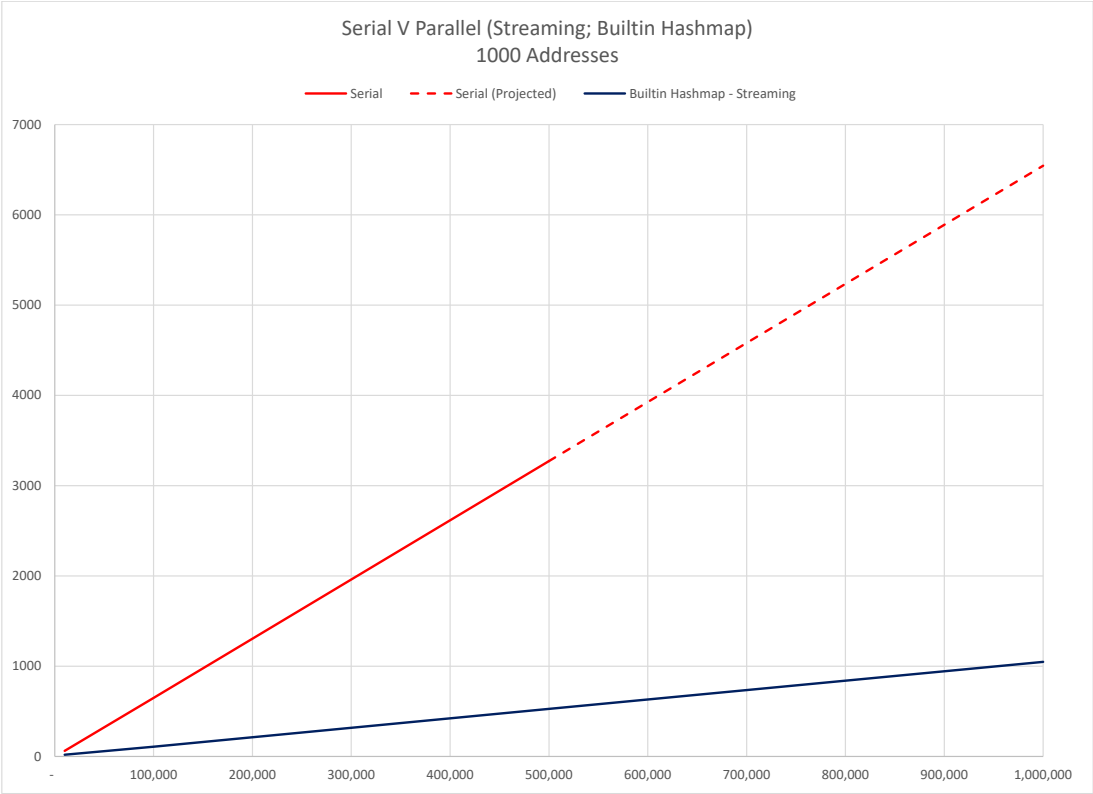

Figure 1: ms vs total packets processed, for the baseline serial implementation and the parallel stream implementation using the builtin hashmap. Because the serial version was so much slower to run, larger values are linearly extrapolated.

3.3 Critical Time vs Overall Time

We examined critical time vs total time to get a sense of the data structure initialization costs of our firewalls. We found that the builtin ConcurrentHashMap and our StripedHashMap took approximately the same amount of time in setup work, while the 2D array took significantly more setup time. This is to be expected, as the HashMaps are initialized empty, while for n addresses, the 2D array needs to initialize $2n$ locks and n^2 booleans. The following chart shows the distribution of differences in between critical time and total time for each data structure, for both threaded and streaming:

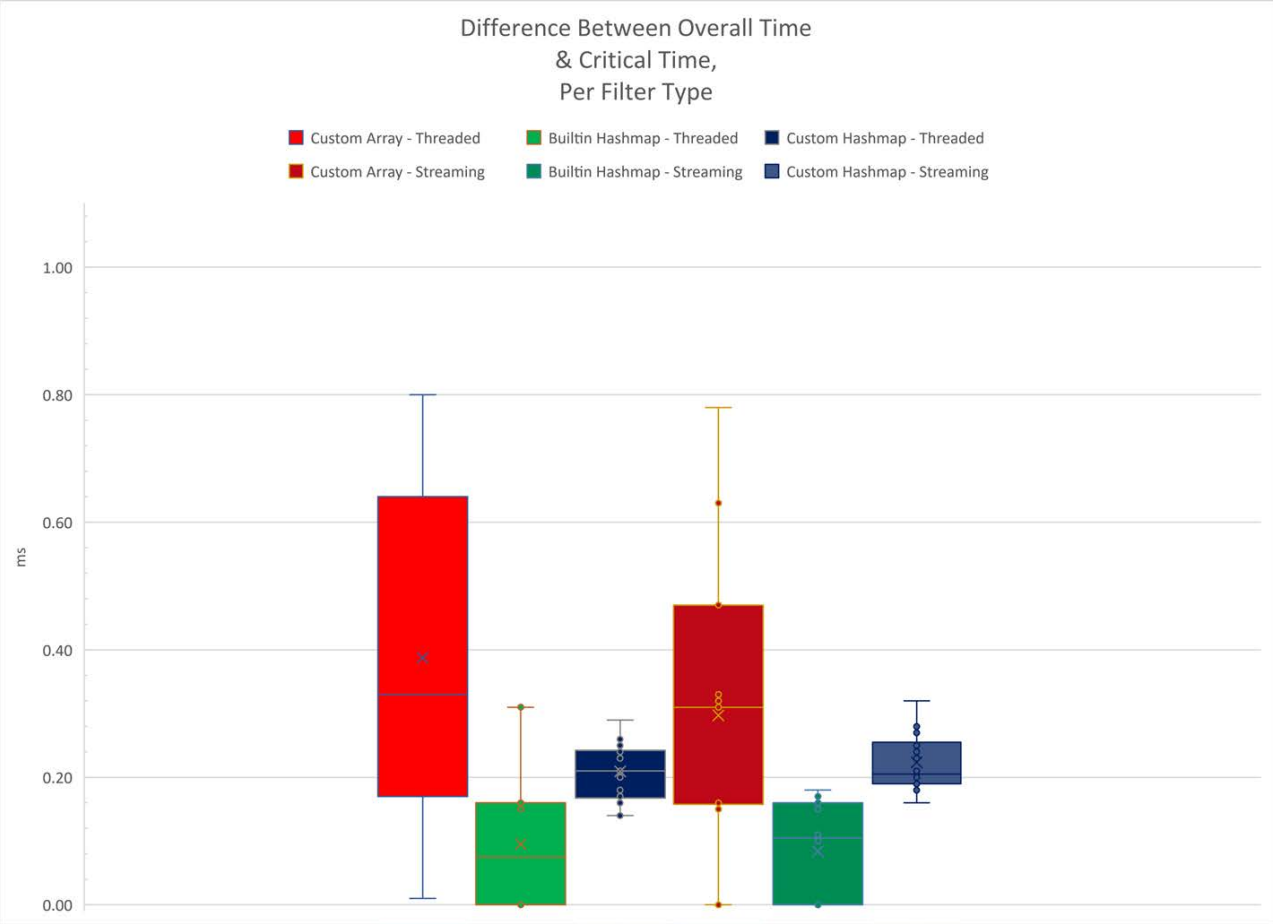

Figure 2: Distribution of differences between overall time and critical time, averaged over 100 iterations

3.4 Threads vs Streams

In general, the streamed packet processing was considerably faster than the threaded packet processing - about 10-15% depending on the underlying data structures.

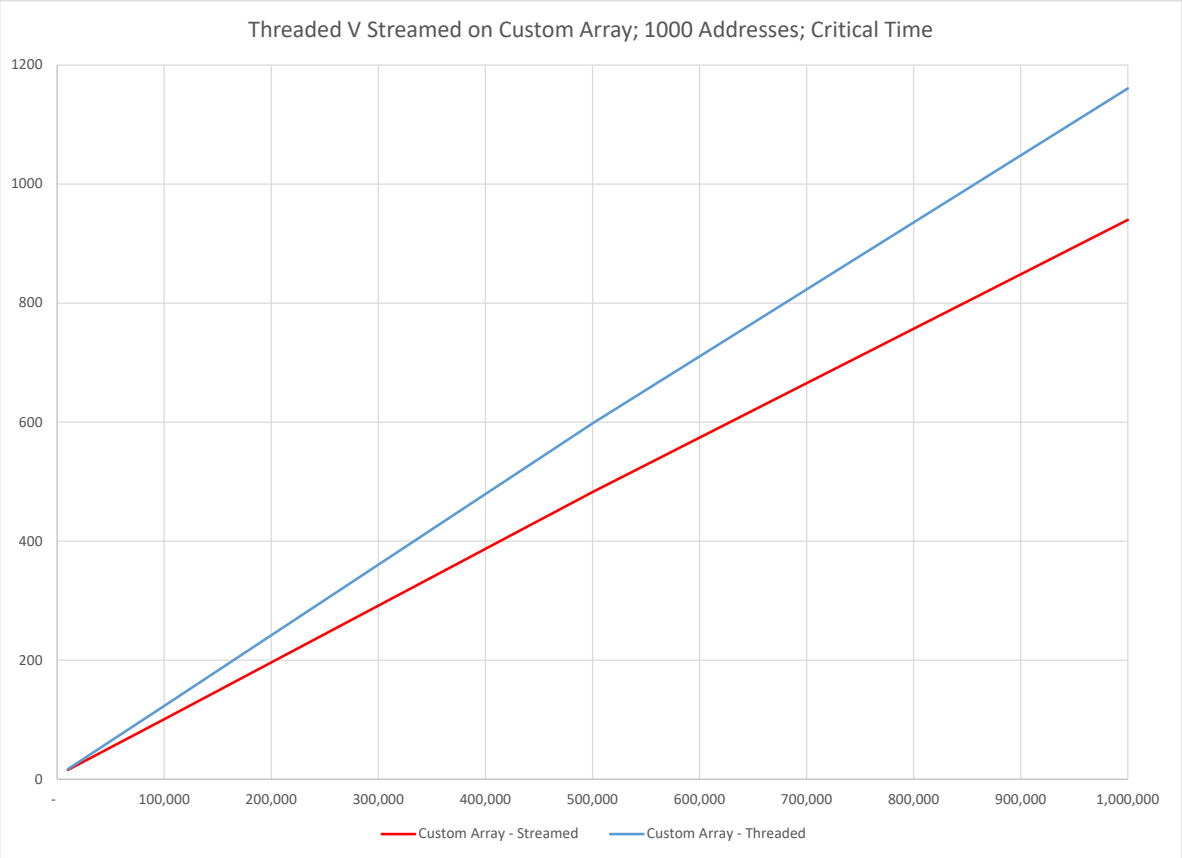

Figure 3: Threaded v streamed critical runtime using our custom array packet filter

3.5 Builtin HashMap vs Striped HashMap vs 2D Array

The performance of our permissions checkers varied widely based on the other parameters. For the threaded packet processing, the builtin ConcurrentHashMap performed the best. For streamed packet processing, the 2D array performed the best. In general, our Striped HashMap was just slightly slower than the ConcurrentHashMap, and the margin was even tighter when run with streams. These results make sense, as our Striped

HashMap is not too different from the ConcurrentHashMap, other than the years of professional optimization.

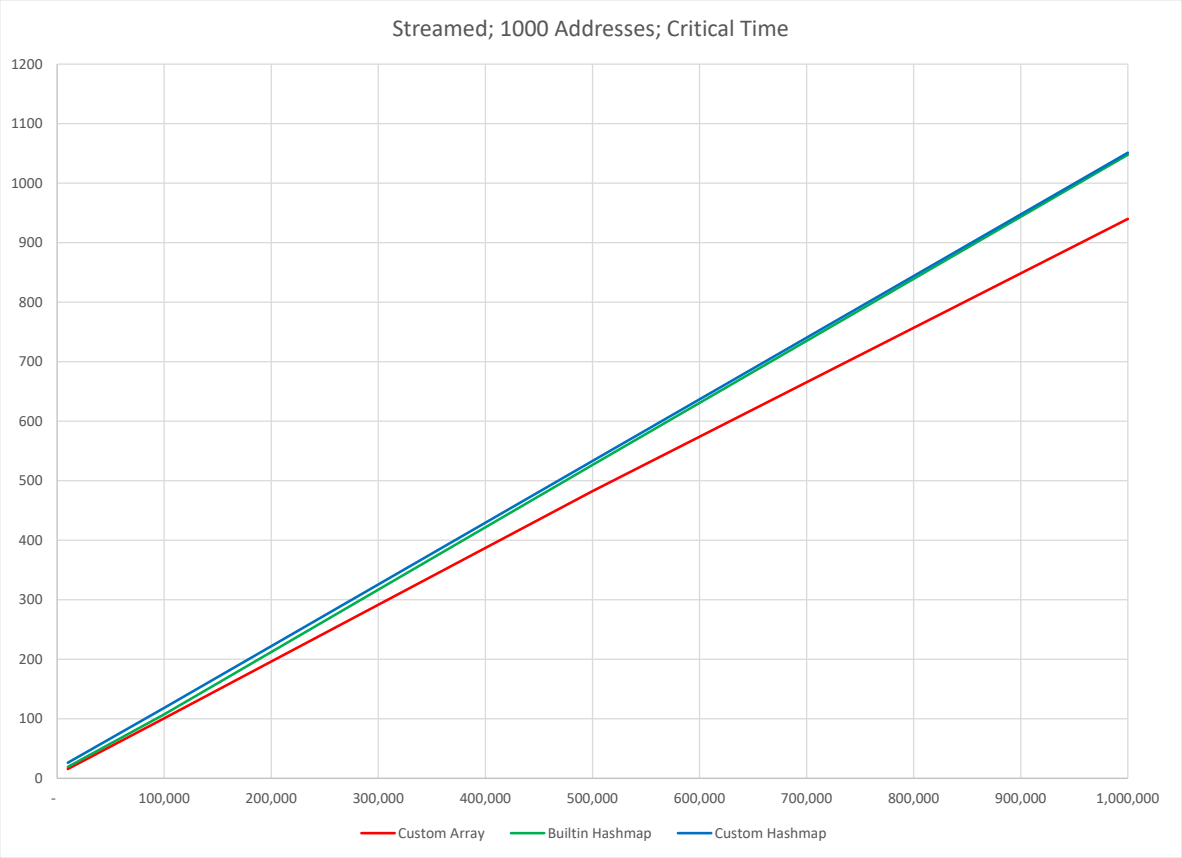

Figure 4: The three data structures' critical runtimes using streams

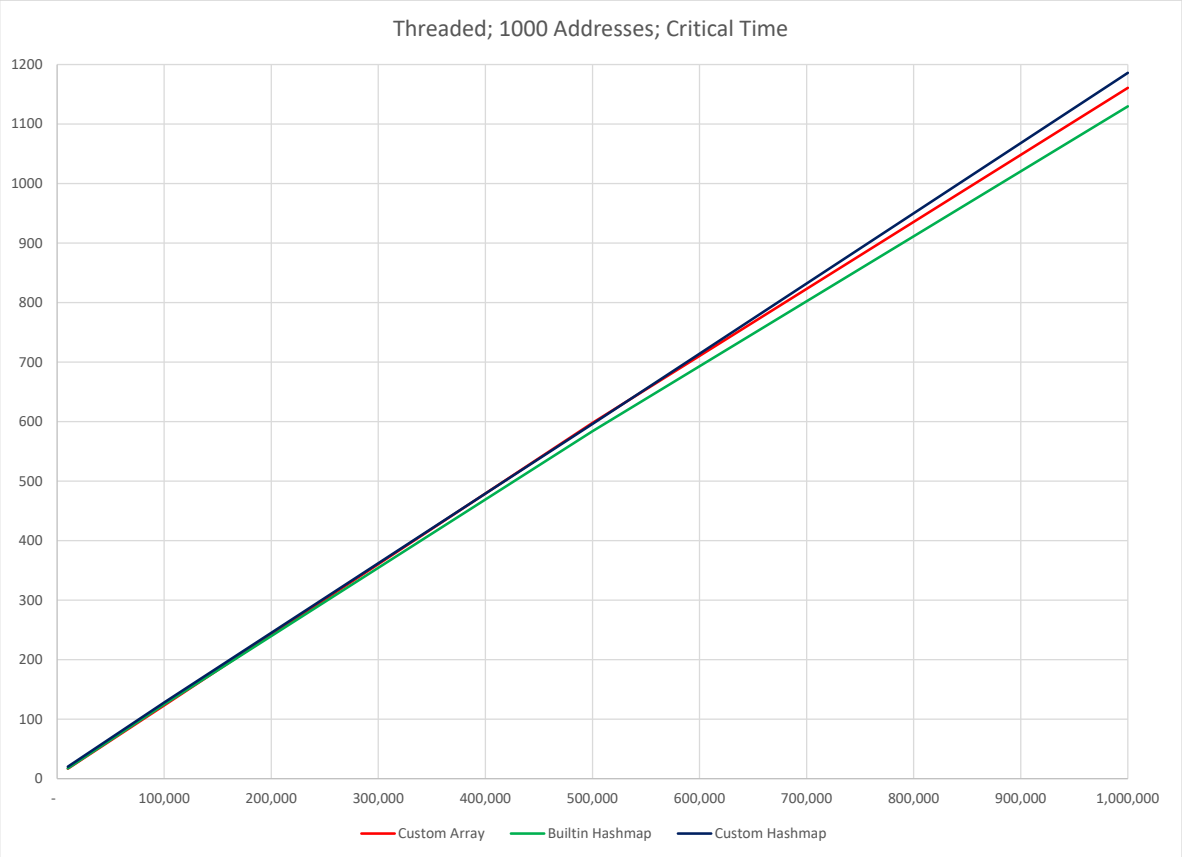

Figure 5: The three data structures' critical runtimes using threads

3.6 Overall Fastest Implementation

Overall, the fastest implementation of our Firewall was streamed packet generation, combined with the 2D array. This is likely a result of the comparatively low stream overhead (initialization time) and the fast access time provided by the 2D array data structure, i.e. the benefit of the space-time tradeoff.

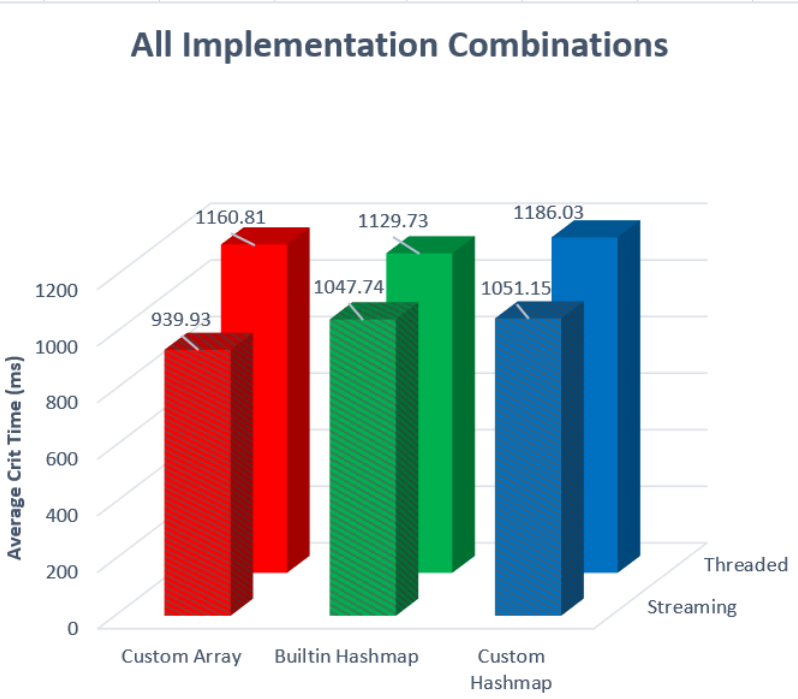

Figure 6: All six implementation combinations, tested with $1,000,000$ packets from $1,000$ addresses

4 Potential Further Investigation

Further topics of investigation might include:

1. Different HashMap implementations. It would be interesting to see if a different concurrent hash map implementation, such as Hopscotch hashing or recursive split order hashing, would provide better performance for our purposes.

2. A fully parallel PacketGenerator. Serial packet generation is a bottleneck on our Firewall. If packets could be generated asynchronously as well as processed asynchronously, we might see higher throughput for packets.

3. Examining the performance of these data structures at much higher computational loads. An astute observer will notice that the critical runtimes shown previously are very slightly concave (there is a slight inflection point at $500,000$ packets). Several hours of computation might be enough to extend these plots far enough to the right to see whether this trend continues (which might be indicative of some logarithmic factor in our workload).

4. Examining the performance of these data structures with many more addresses. At some number of addresses, the space-time tradeoff of using the 2D array will likely swing too far towards space, and the setup time will outweigh the lookup benefits, or the array will not fit in memory at all. Testing the upper bounds of the number of addresses that would fit in the 2D array would be an interesting experiment to see if this data structure would be feasible for a real firewall.

5 Conclusion

In conclusion, we found that for our tests, the best Firewall performance was achieved by streaming packet handling, combined with a 2D array to store address permissions. With the 2D array, we benefit from the constant-time lookups that cannot be achieved with a list of ranges in a traditional HashMap, while not taking too much of a performance hit from the space required to store such an array. We also benefit from low thread contention, as each address has it's own lock, whereas two addresses mapped to the same bucket in a HashMap might share a lock. Java 8's streams performed much better than threads, potentially because Java can better load distribute between threads than we can.

This project demonstrates the difficulty of programming with concurrency. Among the biggest challenges were attempting to ensure correctness, especially writing parallelized tests, and getting a sense of what optimizations and parameter changes were effective in improving performance. It also shows that, while there was functionality that we felt the Java builtins were missing, they are difficult to beat in terms of performance; Java's parallel streams with the ConcurrentHashMap was the second-fastest combination we tested, and was much less memory intensive than the our faster alternative, which is impressive given the extent to which we leveraged memory use and use-case optimizations to achieve our performance.

Notes

[1]Herlihy, Maurice, and Nir Shavit. "1.2 Stream Computing." The Art of Multiprocessor Programming.

[2]Herlihy, Maurice, and Nir Shavit. "13.2.2 A Striped Hash Set." The Art of Multiprocessor Programming.

6 Appendix

6.0.1 Further Visualizations

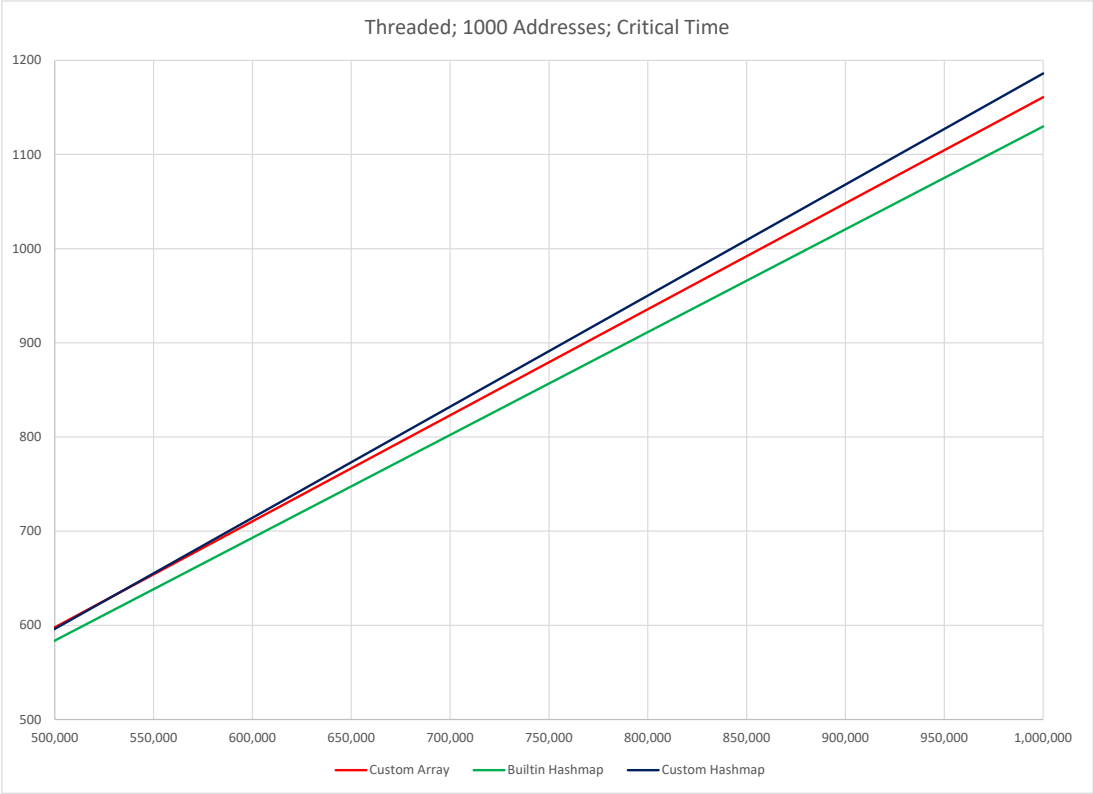

Figure 7: A zoomed view of the three data structures' critical times using threads

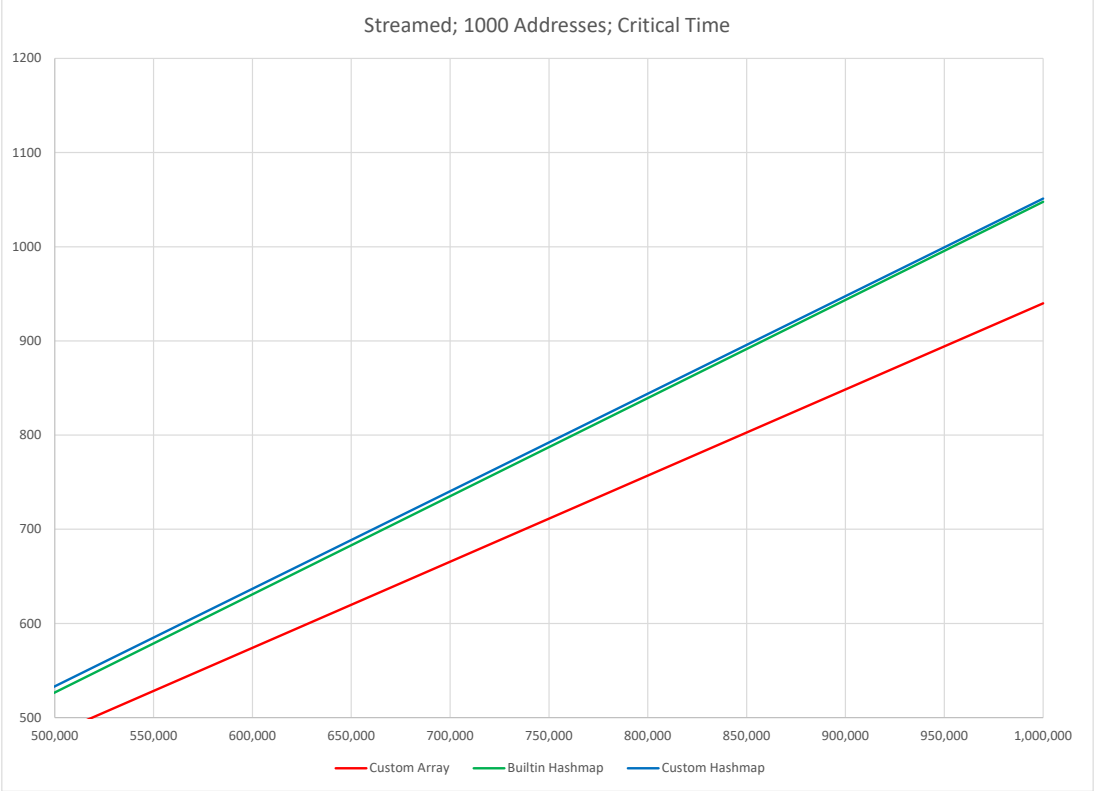

Figure 8: A zoomed view of the three data structures' critical times using streams

6.0.2 Raw Data Follows:

Packet Generation	Packet Filter Type	Number of Packets	of Addresse	Avg. Crit Time (ms)	Avg. Overall Time (ms)
Config 1	Custom Array	1,000	100	2.51	2.84
Config 1	Custom Array	1,000	100	2.2	3
Config 1	Custom Array	10,000	100	16.08	16.72
Config 1	Custom Array	10,000	100	16.1	16.27
Config 1	Custom Array	10,000	1,000	13.96	14.61
Config 1	Custom Array	10,000	1,000	16.85	16.86
Config 1	Custom Array	100,000	100	117.22	117.23
Config 1	Custom Array	100,000	100	116.68	116.85
Config 1	Custom Array	100,000	1,000	124.11	124.74
Config 1	Custom Array	100,000	1,000	123.18	123.51
Config 1	Custom Array	500,000	100	561.64	561.95
Config 1	Custom Array	500,000	100	562.18	562.82
Config 1	Custom Array	500,000	1,000	564.59	564.92
Config 1	Custom Array	500,000	1,000	597.99	598.31
Config 1	Custom Array	1,000,000	100	1109.18	1109.51
Config 1	Custom Array	1,000,000	100	1194.7	1195.2
Config 1	Custom Array	1,000,000	1,000	1155.05	1155.85
Config 1	Custom Array	1,000,000	1,000	1160.81	1160.82
Config 1	Builtin Hashmap	1,000	100	4.53	4.53
Config 1	Builtin Hashmap	1,000	100	4.68	4.99
Config 1	Builtin Hashmap	10,000	100	17.99	18.15
Config 1	Builtin Hashmap	10,000	100	19.01	19.01
Config 1	Builtin Hashmap	10,000	1,000	19.99	20.15
Config 1	Builtin Hashmap	10,000	1,000	17.34	17.34
Config 1	Builtin Hashmap	100,000	100	124.89	124.89
Config 1	Builtin Hashmap	100,000	100	122.84	122.84
Config 1	Builtin Hashmap	100,000	1,000	126.81	126.81
Config 1	Builtin Hashmap	100,000	1,000	124.93	125.08
Config 1	Builtin Hashmap	500,000	100	591.29	591.6
Config 1	Builtin Hashmap	500,000	100	597.29	597.29
Config 1	Builtin Hashmap	500,000	1,000	576.98	576.98
Config 1	Builtin Hashmap	500,000	1,000	583.84	584
Config 1	Builtin Hashmap	1,000,000	100	1136.27	1136.42
Config 1	Builtin Hashmap	1,000,000	100	1143.63	1143.78
Config 1	Builtin Hashmap	1,000,000	1,000	1142	1142.16
Config 1	Builtin Hashmap	1,000,000	1,000	1129.73	1129.73
Config 1	Custom Hashmap	1,000	100	4.84	5.13
Config 1	Custom Hashmap	1,000	100	5.58	5.79
Config 1	Custom Hashmap	10,000	100	21.63	21.89
Config 1	Custom Hashmap	10,000	100	19.79	19.97
Config 1	Custom Hashmap	10,000	1,000	19.56	19.82
Config 1	Custom Hashmap	10,000	1,000	20.17	20.34
Config 1	Custom Hashmap	100,000	100	127.96	128.12
Config 1	Custom Hashmap	100,000	100	126.38	126.58
Config 1	Custom Hashmap	100,000	1,000	129.23	129.43
Config 1	Custom Hashmap	100,000	1,000	127.93	128.16
Config 1	Custom Hashmap	500,000	100	593.63	593.77
Config 1	Custom Hashmap	500,000	100	596.65	596.86
Config 1	Custom Hashmap	500,000	1,000	593.26	593.42
Config 1	Custom Hashmap	500,000	1,000	596.27	596.51
Config 1	Custom Hashmap	1,000,000	100	1173.69	1173.83
Config 1	Custom Hashmap	1,000,000	100	1165.17	1165.42
Config 1	Custom Hashmap	1,000,000	1,000	1181.61	1181.85
Config 1	Custom Hashmap	1,000,000	1,000	1186.03	1186.26

Figure 9: Data table - threaded

Packet Generation	Packet Filter Type	Number of Packets	Number of Addresses	Avg. Crit Time (ms)	Avg. Overall Time (ms)
Config 1	Custom Array	1,000	100	4.24	4.39
Config 1	Custom Array	1,000	100	3.92	4.24
Config 1	Custom Array	10,000	100	14.68	14.84
Config 1	Custom Array	10,000	100	16.53	16.84
Config 1	Custom Array	10,000	1,000	15.13	15.76
Config 1	Custom Array	10,000	1,000	15.64	16.42
Config 1	Custom Array	100,000	100	101.6	101.76
Config 1	Custom Array	100,000	100	100.48	100.63
Config 1	Custom Array	100,000	1,000	103.7	104.03
Config 1	Custom Array	100,000	1,000	100.86	101.33
Config 1	Custom Array	500,000	100	477.17	477.17
Config 1	Custom Array	500,000	100	471.82	471.98
Config 1	Custom Array	500,000	1,000	476.75	477.22
Config 1	Custom Array	500,000	1,000	482.62	483.09
Config 1	Custom Array	1,000,000	100	940.97	940.97
Config 1	Custom Array	1,000,000	100	942.97	943.28
Config 1	Custom Array	1,000,000	1,000	935.89	936.05
Config 1	Custom Array	1,000,000	1,000	939.93	940.25
Config 1	Builtin Hashmap	1,000	100	5.29	5.45
Config 1	Builtin Hashmap	1,000	100	4.2	4.37
Config 1	Builtin Hashmap	10,000	100	19.56	19.72
Config 1	Builtin Hashmap	10,000	100	19.34	19.34
Config 1	Builtin Hashmap	10,000	1,000	22.25	22.4
Config 1	Builtin Hashmap	10,000	1,000	19.08	19.24
Config 1	Builtin Hashmap	100,000	100	125.97	126.13
Config 1	Builtin Hashmap	100,000	100	103.51	103.51
Config 1	Builtin Hashmap	100,000	1,000	107.62	107.62
Config 1	Builtin Hashmap	100,000	1,000	107.53	107.53
Config 1	Builtin Hashmap	500,000	100	585.45	585.45
Config 1	Builtin Hashmap	500,000	100	582.28	582.28
Config 1	Builtin Hashmap	500,000	1,000	526.74	526.74
Config 1	Builtin Hashmap	500,000	1,000	526.74	526.74
Config 1	Builtin Hashmap	1,000,000	100	1156.24	1156.4
Config 1	Builtin Hashmap	1,000,000	100	1112.69	1112.87
Config 1	Builtin Hashmap	1,000,000	1,000	1197.71	1197.81
Config 1	Builtin Hashmap	1,000,000	1,000	1047.74	1047.85
Config 1	Custom Hashmap	1,000	100	5.24	5.52
Config 1	Custom Hashmap	1,000	100	5.54	5.81
Config 1	Custom Hashmap	10,000	100	28.93	29.21
Config 1	Custom Hashmap	10,000	100	26.1	26.35
Config 1	Custom Hashmap	10,000	1,000	28.05	28.37
Config 1	Custom Hashmap	10,000	1,000	26.13	26.37
Config 1	Custom Hashmap	100,000	100	121.95	122.14
Config 1	Custom Hashmap	100,000	100	120.25	120.5
Config 1	Custom Hashmap	100,000	1,000	117.74	117.92
Config 1	Custom Hashmap	100,000	1,000	118.06	118.25
Config 1	Custom Hashmap	500,000	100	547.76	547.95
Config 1	Custom Hashmap	500,000	100	525.48	525.68
Config 1	Custom Hashmap	500,000	1,000	541.97	542.16
Config 1	Custom Hashmap	500,000	1,000	533.19	533.42
Config 1	Custom Hashmap	1,000,000	100	1012.35	1012.55
Config 1	Custom Hashmap	1,000,000	100	1030.8	1031.01
Config 1	Custom Hashmap	1,000,000	1,000	1045.13	1045.29
Config 1	Custom Hashmap	1,000,000	1,000	1051.15	1051.34

Figure 10: Data table - streamed

Packet Generation	Packet Filter	Number of Packets	Number of Addresses	Avg Crit Time (ms)	Avg Overall Time (ms)
Config 1	Serial	1,000	100	7.24	7.33
Config 1	Serial	1,000	100	7.16	7.26
Config 1	Serial	10,000	100	65.74	65.82
Config 1	Serial	10,000	100	65.36	65.42
Config 1	Serial	10,000	1,000	61.14	61.23
Config 1	Serial	10,000	1,000	61.9	61.96
Config 1	Serial	100,000	100	632.25	632.35
Config 1	Serial	100,000	100	643.89	643.96
Config 1	Serial	100,000	1,000	664.02	664.1
Config 1	Serial	100,000	1,000	649.28	649.36
Config 1	Serial	500,000	100	3216.68	3216.75
Config 1	Serial	500,000	100	3214.99	3215.06
Config 1	Serial	500,000	1,000	3299.43	3299.53
Config 1	Serial	500,000	1,000	3272.17	3273.98
Config 1	Serial	1,000,000	100	[redacted]	
Config 1	Serial	1,000,000	100		
Config 1	Serial	1,000,000	1,000		
Config 1	Serial	1,000,000	1,000		

Figure 11: Data table - serial