

# 1 Performance Testing & Evaluation

This section focusses on the comparison testing which was carried out between the C and Java implementations of basic middleboxes. Evaluation of the results will then be discussed and further improvements to the Java solutions will be considered.

In order to fully understand the capabilities of middleboxes, testing was carried out on Imperial College's Large Scale Distributed Systems (LSDS) test-bed. Although this system consists of numerous machines, tests were carried out using just 2. The first machine was used to host the middlebox application and receive the packets. The other machine was used as the client and ran the pktgen (section 1.1) software allowing it to generate packets at up to 10Gbps in order to take advantage of the machines network interface controllers. Each machine had 2 Intel Xeon ES-2690 2.90Ghz processor which when hyper-threaded, resulted in 32 cores utilising 32GB of memory.

## 1.1 Packet Generating

Packet generating is the act of creating packets with random payloads to be sent to certain MAC addresses on the network. This can either be done via the use of specialised hardware or using software. They are used for load testing of packet processing applications to test the amount of data which applications can process per second. This can reveal whether limitations on a system is software or hardware based.

Talk about pktgen module in kernel - does dpdk version use this

Pktgen is open source software tool, maintained by Intel, which aims to generate packets using the DPDK framework. It can generate up to 10Gbits of data per second with varying frame sizes ranging from 64 to 1518 bytes, and send the data in the form of packets across a compatible network interface card/controller. It has a number of benefits which include:

- Real time packet configuration and port control
- Real time metrics on packets sent and received
- Handles UDP, TCP, ARP and more packet headers
- Can be commanded via a Lua script

Pktgen's transmitting performance can be reduced depending on the frame size. This was negated to keep it a constant within the tests by running the application with multiple lcores on the same port with different queues, which allow the transmitting speed to match those of the NIC. With the packet size varying, obviously the number of packets transmitted per second is dependant on this. The differences are shown in Table 1.

| Packet Size | Packet/s (millions) | MBit/s |
|-------------|---------------------|--------|
| 64 (min)    | 14.9                | 9999   |
| 128         | 8.4                 | 9999   |
| 256         | 4.5                 | 9999   |
| 512         | 2.3                 | 9999   |
| 1024        | 1.2                 | 9999   |
| 1518 (max)  | 0.8                 | 9999   |

Table 1: Pktgen performance for varying packet sizes transmitting in 32 packet burst.

mention  
constants

## 1.2 C Packet Capture performance test

The first test on the LSDS test-bed involved comparing the C and Java implementations of a packet capturing application which simply received the packets and freed them straight away without any further transmission. The C implementation is used to give the optimal readings possible from this and further tests, since very limited processing is carried out between receiving and dropping the packet. Figure 1 shows the performance of the application at varying packet sizes.

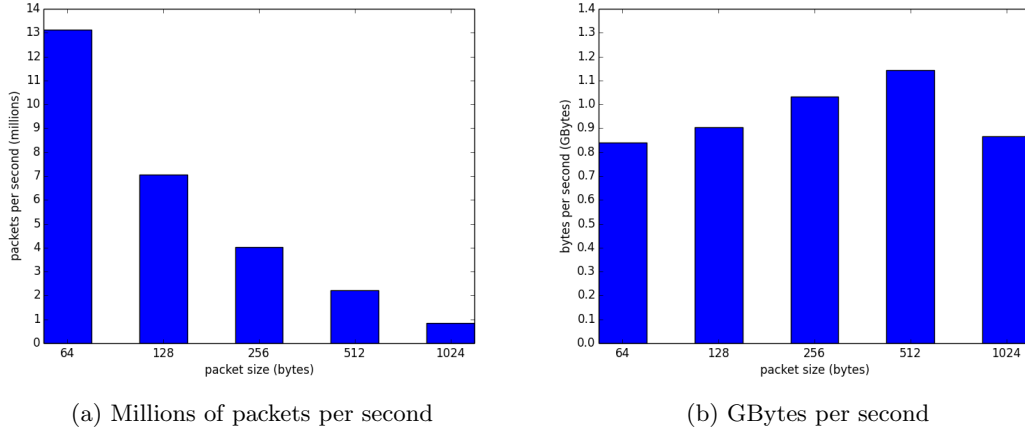


Figure 1: Performance results of C implementation of packet capture at varying transmission packet sizes

Figure 1a shows the expected results as the packet size increases the number of packets processed per seconds decreases. This is obvious since the NIC has a maximum limit which it can process data per second therefore an inverse correlation between packet size and number of packets exists. If further calculation are carried out, by multiplying the packet size by the number of packets the values turn out to be around the 1000 million bytes per second (1 GBps). This results in roughly 8 Gbps which turns out to be the maximum processing performance of 1 core.

The results of figure 1b should theoretically have equal bytes per second readings due to the limit of the NIC or the software. However, the results are a lot more skewed than expected and this can be accredited to the memory management of the DPDK framework depending the packet size. Up to a certain limit between 512 bytes and 1024 bytes, DPDK starts splitting packets into multiple segments for storage. This value is determined on a few factors such as the size of the mbuf struct, headroom and pointer sizes.

Although this first test was simple, it showed a few characteristics of fast packet processing. Generally the hardware is more powerful than the software, therefore requiring more than 1 core to handle the received packets of a NIC. Even so, 1 core testing prevails throughout the next tests as to compare with the C results.

### 1.3 Java Packet Capture performance test

The same testing was carried out with an identical packet capture algorithm which was instead implemented in Java using the DPDKJava framework described in section ???. All testing was done using 64 byte frames to make sure a consistency was kept throughout. Frame size of 64 bytes was chosen as this size is the hardest for applications to process simply because of the sheer number of packets per second which are received. This is an ultimate test of an applications packet throughput, and generally if it can handle packets at 64 bytes, it can handle much larger packets since the packet rate will be reduced. These results are shown in figure ??.

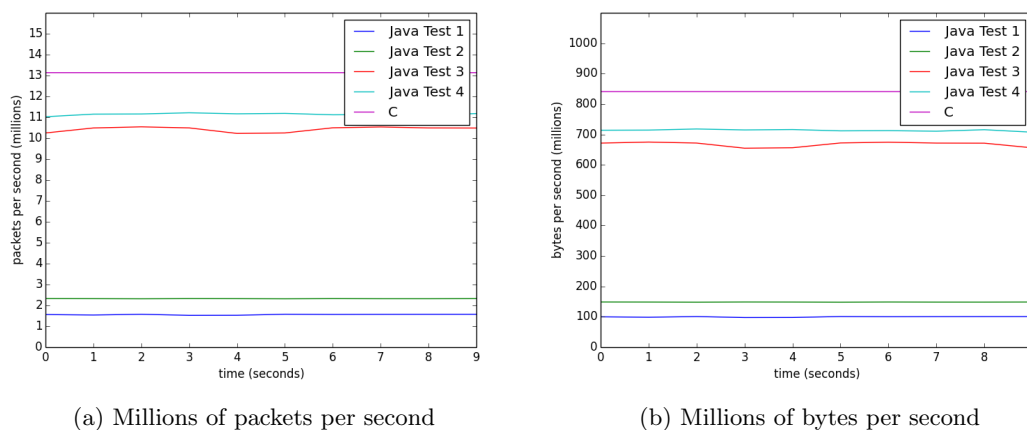


Figure 2: Performance results of C vs Java implementations of packet capture application for all rounds of testing

Figures 2a and 2b indicate that after the first testing using the initial DPDKJava framework that the packet and data throughput was extremely low compared to the C version. Obviously these results are extremely closely related to each other resulting in the C version been roughly 8 times quicker after the first test. Although the native implementation was expected to be

different  
frame  
sizes?

To check where the problems lied within the code, a Java profiler (JProfiler) was used to check multiple parts of the code including memory usage, CPU usage and the number of method calls and the average time per method call. This provided invaluable analysis of where the problems were, although the profiler significantly reduced the performance of the application since it connects to the JVM and reads the data itself. Even so, obvious bottlenecks could be spotted allowing for performance improvement implementations to be made.

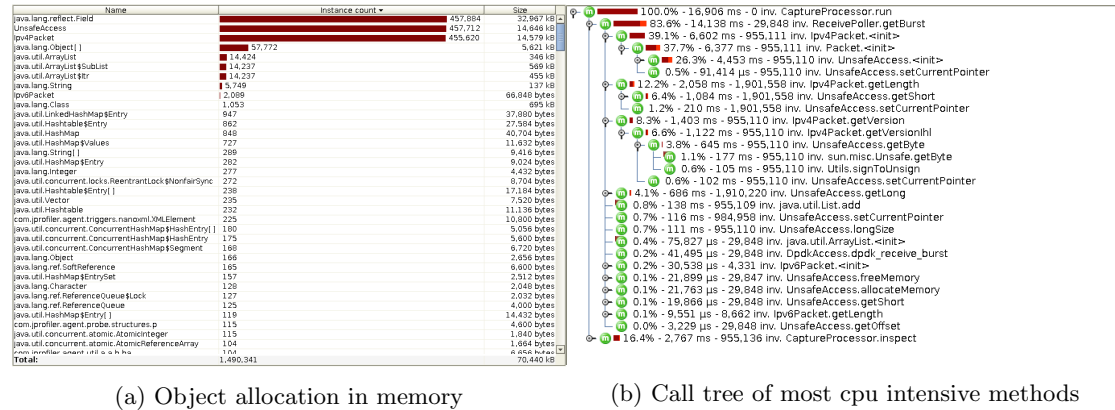


Figure 3: Profiler output when analysing the first test of packet capture

Figure 3 shows some of the output from the profiler which indicated where the main problems were in terms of memory usage and performance. From this, a number of performance upgrades were implemented:

**Capture Processor fields** The first improvement were to the Packet Capture application itself. The associated processor originally stored the ReceivePoller and PacketFreeer objects within a list to allow for easy iteration if there were numerous objects. Since the Packet Capture application only used 1 of each, separate fields could be used for the objects which removed the list accessing times and iterations.

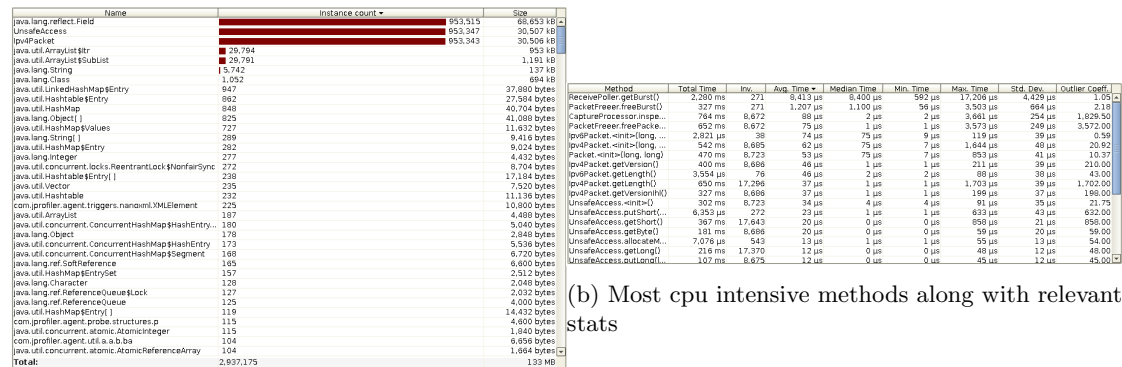
**Object lifespan** The other improvements were made to the actual DPDKJava framework. These involved utilising objects throughout the application lifespan instead of creating new ones on every loop. This dramatically reduced the number of initializing methods invoked for the objects and reduced the memory usage in the heap, which reduced the number of times the Java garbage collector was invoked. The class which caused the most problem with this was the ArrayList, which were created on every loop to pass packets through the Java system. Since the framework uses threads without the need to synchronise objects, an ArrayList could be created on initialisation and simply cleared before been used again.

**Off heap allocation** Finally, instead of allocating new off heap memory to receive the packet pointers through on every loop, a memory bank was allocated on initialisation and the same

memory was simply overwritten on every loop iteration. This stopped expensive calls to the allocate and free methods.

Again, the same tests were carried out to check the performance increase of the new framework implementation. Shown in figure 2 by comparing the blue and green lines, there was a slight performance increase of roughly 1 millions packets per second just by these simple changes, although it was still significantly down on performance to the C version.

Further profiling of the application was undertaken on the improved application. Figure 4 shows the analysis which outlined 2 further major improvements which could be made. Each one of these improvements were undertaken and performance tested individual.



(a) Object allocation in memory

(b) Most cpu intensive methods along with relevant stats

Figure 4: Profiler output when analysing the first test of packet capture

**Packet creation** Of course, for every packet entering the application a packet object is created for easy referencing further down the application pipe-line. However, not using packet objects would significantly reduce the usability and scalability of applications. It was decided not to alter this. However, for each packet initialization a new UnsafeAccess object was been created for use with accessing packet header information. However, since each thread controls its own packets and therefore can only process 1 packet at a time, 1 UnsafeAccess object could be shared between all packets which significantly reduced the number of objects on the heap. This improvement created a vast performance increase, rising the number from 2.4 million to just over 10 million packets per second as indicated with the red line in figure 2.

**Send and Free lists** Further problems were identified with the lists of packets awaiting to be sent and freed. This list was been iterated over with the Packet's mbuf pointer then been stored in an off heap memory bank waiting to be freed. This was pointless since the packets mbuf pointer could directly be put into the off heap memory, therefore eliminating the need for the list while also allowing the packet objects to be dereference quicker. Again, this bottleneck was fixed and resulted in a further improvement of roughly 1 million packets per second as indicated with the turquoise line in figure 2.

The results show that the performance was further improved and pretty close to that of the C implementation. After further profiling, there was only 1 obvious improvement which could

be made, This would be to replace the list storing packets received from the poller. However, replacing this with a custom implementation using off heap memory would firstly reduce the usability of the code and would also mean that the code was drifting away from the Java language. It was decided not to implement this fix and instead to continue with testing of other applications, assuming that the would be negated by applications. This initial series of testing also proved that a dramatic increase in performance can be achieved simply by programming the applications efficiently, by trying to reduce the number of objects created.

## **1.4 Firewall performance test**

Since no further improvements to the DPDKJava framework could be undertaken, further analysis of its capabilities was undertaken via the use of a firewall implementation. This implementation actually did some processing of the packets by accessing header information and checking the data against set rules about source IP address and protocol type. If the checks were passed, the packets are forwarded otherwise they are dropped and freed from memory. This provides a suitable test to compare how Java handles repeated processing of different objects in a very short time frame compared to the native implementation.

The tests were carried out for a range of different packet sizes so not to limit the conclusions which could be drawn from the results.

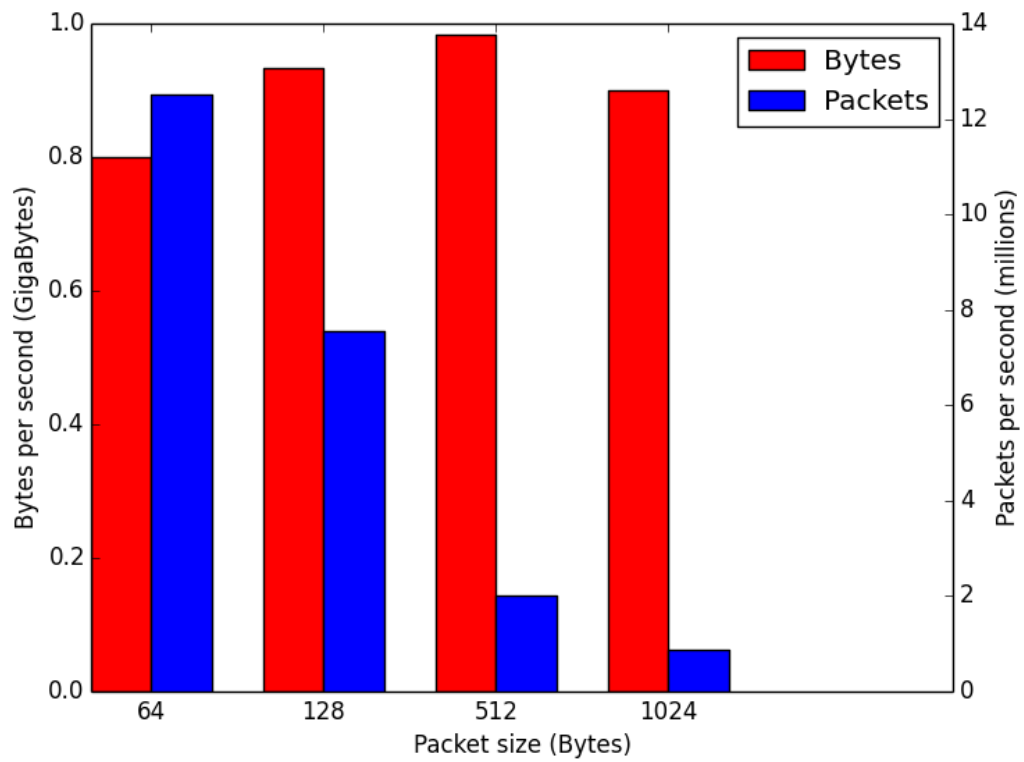


Figure 5: Firewall C implementation performance test results

The results from the C implementation of the firewall are shown in 5 which roughly follows the same trend from the packet capture results. The number of bytes processed per seconds roughly stays the same while the number of packets reduces as the packet size increases. Compared to the packet capture program, it has reduced byte throughput as expected due to the increased packet processing of the firewall algorithm, therefore the number of packets processed per second is slightly reduced as well.

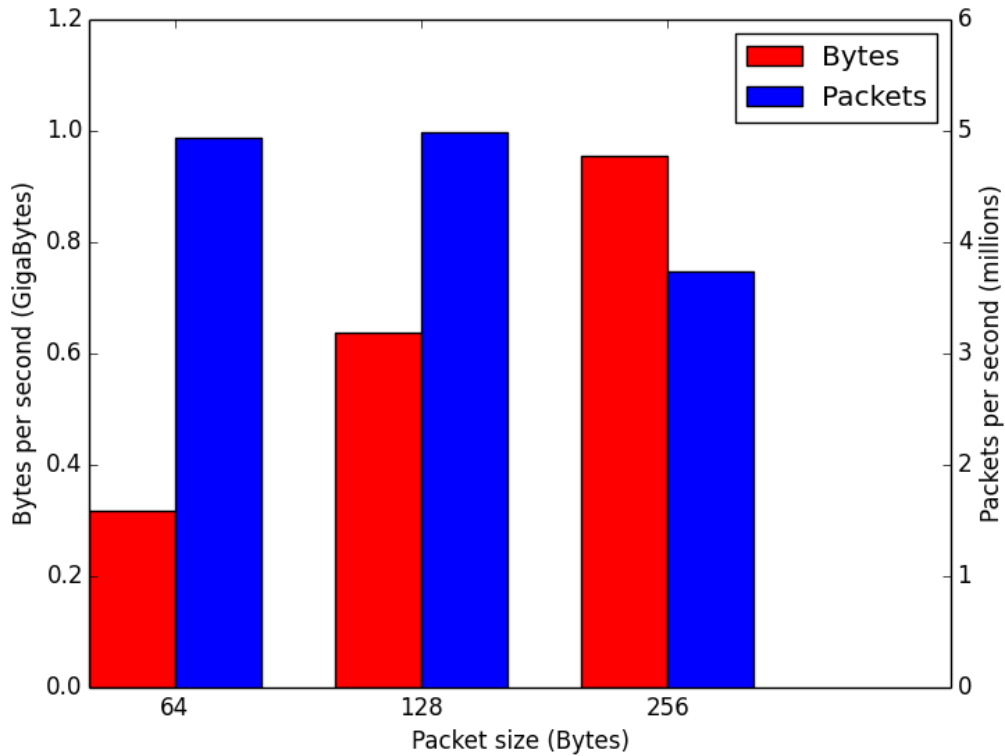


Figure 6: Firewall Java implementation performance test results

Figure 6 shows the performance of the Java implementation of a firewall. Both the packets per second and bytes per second are considerably lower than those in the C implementation. Considering the previous results for the packet capture algorithm, the only explanation for this is the extra processing per packet inspection and the transmission of packets. The packet sizes used for the performance tests were different to those used for the C test. This is because packet sizes of 512 and 1024 bytes actually stop the application from polling the packets. Instead, the NIC was dropping all of the packets as none of them were been removed from the queues. The likely cause of this is the initial configuration which all of the applications adhere to. However, this did mean that no data for packets larger than 256 bytes could be gathered for evaluation.

Using the Java profiling tool initially brought up a few minor bottlenecks which were solved before the results were taken. Afterwards, no bottlenecks could be found which were fixable and the analysis simply eluded to the inspection of the packets and the iteration over the firewall rules as the major problem.

This led to the conclusion that the only possible way to match the C performance was to increase the number of threads and receive queues per port. The idea been that performance should scale with extra threads polling packets from their own receive queues. Initial testing on this theory was carried out trying to increment the number of threads and test the performance increase. The expectation was that eventually the performance would closely relate to that of the hardware.



However, problems with the application resulted in very inconsistent readings from the statistic profiler which suggested there was a problem with the affinity threads. Time constraints resulted in no solution to this issue and therefore no meaningful data was collected.

## **1.5 Final Evaluation**

Even though the DPDKJava framework abstracts a lot of work away from the user and allows for easy programming of applications, the framework still doesn't perform to the same standard as native implementations. The performance testing on the firewall application proved this and without tests for the threaded applications no guarantee of performance matching can be made. Theoretically the performance difference can be made via adding extra threads, but extra CPU loading would then have to be taken into consideration.

This came as a real disappointment since the design consideration considered all aspects of problems within the framework and initial testing with the packet capture application looked promising.