1 DPDK

This section will go into much more detail about the Data Plane Development Kit (DPDK), focussing on the basic concepts used by the fast packet processing framework for use within custom applications.

1.1 Environment Abstraction Layer (EAL)

The EAL abstracts the specific environment from the user and provides a constant interface in order for applications to be ported to other environments without problems. The EAL may change depending on the architecture (32-bit or 64-bit), operating system, compilers in use or network interface cards. This is done via the creation of a set of libraries that are environment specific and are responsible for the low-level operations gaining access to hardware and memory resources.

On the start-up of an application, the EAL is responsible for finding PCI information about devices and addresses via the igb_uio kernel module, performing physical memory allocation using huge pages and starting user-level threads for the logical cores (lcores) of the application.

reference this

1.2 huge pages

Huge pages are a way of increasing performance when dealing with large amounts of memory. Normally, memory is managed in 4096 byte blocks known as pages, which are listed within the CPU memory management unit (MMU). However, if a system uses a large amount of memory, increasing the number of standard pages is expensive on the CPU as the MMU can only handle thousands of pages and not millions.

The solution is to increase the page size from 4KB to 2MB or 1GB (if supported) which keeps the number of pages references small in the MMU but increases the overall memory for the system. Huge pages should be assigned at boot time for better overall management, but can be manual assigned on initial boot up if required.

DPDK makes uses of huge pages simply for increased performance due to the large amount of packets in memory. This is even the case if the system memory size is relatively small and the application isn't processing an extreme number of packets.

1.3 Ring Buffer

A ring buffer is used by DPDK to manage transmit and receive queues for each network port on the system. This fixed sized first-in-first-out system allows multiple objects to be enqueued and dequeued from the ring at the same time from any number of consumers or producers. A major disadvantage of this is that once the ring is full (more of a concern in receive queues), it no more objects can be added to the ring, resulting in dropped packets. It is therefore imperative that applications can processes packets at the required rate.

1.4 Memory usage - malloc, mempool, mbuf

DPDK can make use of non-uniform memory access (NUMA) if the system supports it. NUMA is a method for speeding up memory access when multiple processors are trying to access the same memory and therefore reduces the processors waiting time. Each processor will receive its own bank of memory which is faster to access as it doesn't have to wait. As applications become more extensive, processors may need to share memory, which is possible via moving the data between memory banks. This somewhat negates the need for NUMA, but NUMA can be very effective depending on the application. DPDK can make extensive use of NUMA as each logical core is generally responsible for its own queues, and since queues can't be shared between logical cores, data sharing is rare.

As discussed previously , DPDK uses hugepages in memory and therefore it provides its own malloc library, which as expected, allocates hugepage memory to the user. However, as DPDK focusses on raw speed, the use of the DPDK malloc isn't suggested as pool-bases allocation is faster. DPDK also provides ways to malloc memory on specific sockets depending on the NUMA configuration of the application.

ref

finish this section

1.5 Poll Mode Driver (PMD)

A Poll Mode Driver (PMD) is responsible

2 Initial language comparison

Before any implementation or specific design considerations were undertaken, an evaluation of the performance of C, Java and Java using the Java Native Interface (JNI) was carried out. Although data from existing articles and websites could be used for Java and C, there was no existing direct comparisons between them and the JNI, therefore custom tests were carried out.

The JNI is inherently seen as a bottleneck of an application (even after its vast update in Java 7).

article on this

reasons why JNI is slow

As this application would be forced to use the JNI, numeric values of its performance was helpful to evaluate the bridge in speed required to be overcome.

2.1 Benchmarking Algorithm

As discussed previously , there are always advantages and disadvantages of any algorithm used for benchmarking. In order to minimise the disadvantages, an algorithm was used which tried to mimic the procedures which would be used in the real application, just without the complications. Algorithm **??** shows that the program basically creates 100,000 packets individually and populates their fields with random data, which is then processed and return in the 'result' field. This simulates retrieving low-level packet data, interpreting and acting upon the data, and then setting data within the raw packet.

ref this

Algorithm 1 Language Benchmark Algorithm

```
function MAIN
    for i = 1 to 100000 do
        $p \leftarrow$ *Initialise Packet*
        POPPACKET(p)
        PROPACKET(p)

function POPPACKET(Packet p)                          ▷ Set data in a packet
    $p.a \leftarrow randomInt()$
    $p.b \leftarrow randomInt()$
    $p.c \leftarrow randomInt()$
    $p.d \leftarrow randomInt()$
    $p.e \leftarrow randomInt()$

function PROPACKET(Packet p)                          ▷ Process a packet
    $res \leftarrow p.a * p.b * p.c * p.d * p.e$
    $p.result \leftarrow res$
```

For the JNI version, the same algorithm was used, however, the PopPacket method was carried out on the native side to simulate retrieving raw packet data. The ProPacket method was executed on the Java side with the result been passed back to the native side.

2.2 Results

Each language had the algorithm run 1000 times in order to minimise any variations due to external factors. Figures show that C was considerably quicker than Java, while Java using the JNI was extremely slow.

ref this

expand on this

2.3 Further Investigation

Due to the very poor performance of the JNI compared to other languages, further investigations were carried out to find more specific results surrounding the JNI.

Is this relevant

3 Design Considerations

3.1 Data Sharing

The proposed application will be sharing data between the DPDK code written in C and the Java side used for the high functionality part of the application. This requires a large amount of data, most noticeably packets, to be transferred between 'sides' in a small amount of time.

Diagram of packets from NIC using c through 'technique' and then processing packets in java and then back

A few techniques for this are available with Java and C, all with different performances and ease-of-use.

3.1.1 Objects and JNI - using heap and lots of jni calls

By far the simplest technique available is using the Java Native Interface (JNI) in order to interact with native code and then retrieve the required via this. This can be done 2 ways, either by creating the object and passing it as a parameter to the native methods or creating an object on the native side via the Java environment parameter. Both ways require the population of the fields to be done on the native side. From then on, any data manipulation and processing could be done on the Java side. Unfortunately, this does require all data to be taken from the object and placed back into the structs before packets can be forwarded. Obviously this results in a lot of unneeded data copying, while the actual JNI calls can significantly reduce the speed of the application as shown in . ref this

3.1.2 ByteBuffers - Non-heap and heap memory

ByteBuffers are a Java class which allow for memory to be allocated on the Java heap (non direct) or outside of the JVM (direct). Non direct ByteBuffer's are simply a wrapper for a byte array on the heap and are generally used as they allow easier access to bytes, as well as other primitive data types.

Direct ByteBuffers allocate memory outside of the JVM in native memory. This firstly means that the only limit on the size of ByteBuffers is memory itself. Furthermore, the Java garbage collector doesn't have access to this memory. Direct ByteBuffers have increased performance since the JVM isn't slowed down by the garbage collector and intrinsic native methods can be called on the memory for faster data access.

3.1.3 Java Unsafe - non-heap

The Java Unsafe class is actually only used internally by Java for its memory management. It generally shouldn't be used within Java since it makes a safe programming language like Java an unsafe language (hence the name) since memory access exceptions can be thrown. It can be used for a number of things such as:

- Object initialisation skipping
- Intentional memory corruption
- Nullifying unwanted objects
- Multiple inheritance
- Dynamic classes
- Very fast serialization

Obviously without proper precautions any of these actions can be dangerous and can result in crashing the full JVM. This is why the Unsafe class has a private constructor and calling the static Unsage.getUnsafe() will throw a security exception for untrusted code which is hard to bypass. Fortunately, Unsafe has its own instance called 'theUnsafe' which can be accessed by using Java reflection : ref this

```
Field f = Unsafe.class.getDeclaredField("theUnsafe");
f.setAccessible(true);
Unsafe unsafe = (Unsafe) f.get(null);
```

Code 1: Accessing Java Unsafe

Using Unsafe then allows direct native memory access to retrieve data in any of the primitive data formats. Custom objects with a set structure can then be created, accessed and altered using Unsafe which provides a vast increase in performance over traditional objects stored on the heap. This is mainly thanks to the JIT compiler which can use machine code more efficiently.

3.1.4 Evaluation

3.1.5 JNA?

3.1.6 Packing C Structs

Structs are a way of defining complex data into a grouped set in order to make this data easier to access and reference as shown in Code **??**.

```c
struct example {
    char *p;
    char c;
    long x;
    char y[50];
    int z;
};
```

Code 2: Example C Struct

On modern processors all commercially available C compilers will arrange basic C datatypes in a constrained order to make memory access faster. This has 2 effects on the program. Firstly, all structs will actually have a memory size larger than the combined size of the datatypes in the struct as a result of padding. However, this generally is a benefit to most consumers as this memory alignment results in a faster performance when accessing the data.

Explain why it has faster performance

Nested padding in struct?

C struct field always in given order

Inconsistencies with datatype length so using uint32t etc

Code **??** shows a struct which has compiler inserted padding. Any user wouldn't know the padding was there and wouldn't be able to access the data in the bits of the padding through conventional C dereferencing paradigm. This example does assume use on a 64-bit machine with 8 byte alignment, but 32-bit machines or a different compiler may have different alignment rules.

```c
struct example {
    char *p;        // 8 bytes
    char c;         // 1 byte
    char pad[7];    // 7 byte padding
    short x;        // 2 bytes
    char pad[6];    // 6 byte padding
    char y[50];     // 50 bytes
    int z;          // 4 bytes
};
```

Code 3: Example C Struct with compiler inserted padding

Mention this is on 64-bit machine and obviously you don't notice padding

Example making sure compiler doesn't pad

Since the proposed application in this report requires high throughput of data, the initial thought would be that this optimisation is a benefit to the program. Generally this is the case, but for data which is likely to be shared between the C side and Java side a large amount, data accessing is far quicker on the Java side if the struct is padded. This results in certain structs been forced to be padded when compiled, more noticeably, those used for packet headers.

Proof on speed

3.1.7 Javolution

3.1.8 Performance testing

4 General Implementation

4.1 Shared Libraries

4.2 Handling Errors