1 Initial language comparison

Before any implementation or specific design considerations were undertaken, an evaluation of the performance of C, Java and 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

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.

reasons why JNI is slow

1.1 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 1 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

```
1:  function Main
2:      for i = 1 to 100000 do
3:          p ← Initialise Packet
4:          PopPacket(p)
5:          ProPacket(p)

6:  function PopPacket(Packet p)                    ▷ Set data in a packet
7:      p.a ← randomInt()
8:      p.b ← randomInt()
9:      p.c ← randomInt()
10:     p.d ← randomInt()
11:     p.e ← randomInt()

12: function ProPacket(Packet p)                    ▷ Process a packet
13:     res ← p.a * p.b * p.c * p.d * p.e
14:     p.result ← 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 to be entered back into the packet structure.

Timing within the algorithm for all variations was carried out between each iteration. This firstly eliminated any initial start-up time associated with the application which is common with the JVM. Secondly, any calls for time stamps to the system would be miminised as 100,000 iterations would occur in-between them.

1.2 Results

Each language had the algorithm run 1,000 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

1.3 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

2 Design Considerations

2.1 Data Sharing

The proposed application will be sharing data between the DPDK code written in C (low level) and the Java (medium level) side used for the highly abstracted 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.

2.1.1 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

2.1.2 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.

2.1.3 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 This is where any instance of an object can be created from the class, but no constructors are used meaning that the object created without any of the fields set. This has a number of uses including security check bypassing, creating instances of objects which don't have a public constructor and allowing multiple objects of a singleton class.

Intentional memory corruption This allows the setting of private fields of any object. It is a common way of bypassing security features as private fields to allow access to certain situation can be overwritten to gain access.

Nullifying unwanted objects This has a common use of nullifying passwords after they have been stored as strings. If a password is stored as a string in Java, even setting the field to (null) will only dereference it. The original string will still be in memory after the dereferencing up until it is garbage collected. Even rewriting the field with a new field won't work as strings are immutable in Java. This makes it susceptible to a timing attack to retrieve the password. Using unsafe allows for the actual memory location to be overwritten with random values to prevent this.

Multiple inheritance Java doesn't allow multiple inheritance within its class declaration of casting. However, using Unsafe any object can be cast to any other object without a compiler of run-time error. This obviously only works if data fields are compliant with each other and any method invocations are referenced.

Very fast serialization The Java *Serializable* abstraction is well known to be slow, which can be a major bottleneck in fast processing applications over a network. Even the *Externalizable* functionality isn't much factor and that requires an class schema to be defined. However, custom serializing can be extremely fast using the Unsafe Class. Basically allocating memory and then putting/getting data from the memory requires little JVM usage and can be done with machine instructions.

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 (off heap) 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. This also removes the need for copying of data between memory locations, structs and objects, therefore meaning it is zero-copy.

2.1.4 Evaluation

2.1.5 JNA?

2.1.6 Packing C Structs - in its own section?

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 2.

```
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 4 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 (only via pointer arithmetic). 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.

```
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 and order of elements can pay an important part in this

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 packed (no padding). This results in certain structs been forced to be packed when compiled, more noticeably, those used for packet headers.

Proof on speed

Packed structures mean there are no gaps between elements, required alignment is set to 1 byte. Also __attribute__((packed)) definition means that compiler will deal with accessing members which may get misaligned due to 1 byte alignment and packing so reading and writing is correct. However, compilers will only deal with this misalignment if structs are accessed via direct access. Using a pointer to a packed struct member (and therefore pointer arithmetic) can result in the wrong value for the dereferenced pointer. This is since certain members may not be aligned to 1 byte. In the below example, unint32 is 4 byte aligned and therefore it is possible for a pointer to it to expect 4 byte alignment therefore resulting in the wrong results.

```
#include <stdio.h>
#include <inttypes.h>
#include <arpa/inet.h>

struct packet {
    uint8_t x;
```

```
7       uint32_t y;
8   } __attribute__((packed));
9
10  int main ()
11  {
12      uint8_t bytes[5] = {1, 0, 0, 0, 2};
13      struct packet *p = (struct packet *)bytes;
14
15      // compiler handles misalignment because it knows that
16      // "struct packet" is packed
17      printf("y=%"PRIX32", ", ntohl(p->y));
18
19      // compiler does not handle misalignment - py does not inherit
20      // the packed attribute
21      uint32_t *py = &p->y;
22      printf("*py=%"PRIX32"\n", ntohl(*py));
23      return 0;
24  }
```

Code 4: Example C Struct with compiler inserted padding

On an x86 system (which does not enforce memory access alignment), this will produce

y=2, *py=2

as expected. On the other hand on my ARM Linux board, for example, it produced the seemingly wrong result

y=2, *py=1

However, since a packed struct is much easier to traverse from Java than a padded struct, the decision was made to make certain structs packed within the DPDK framework and then recompile the libraries. This decision could be made since other structs within the DPDK framework were also packed and therefore consideration of this was already made.

Note that if a struct contains another struct, that struct should be packed recursively as-well to ensure the first struct has no padding at all.

Char doesn't have alignment and can start on any address. But 2-byte shorts must start on an even address, 4-byte ints or floats must start on an address divisible by 4, and 8-byte longs or doubles must start on an address divisible by 8. Signed or unsigned makes no difference.

Self-alignment makes access faster because it facilitates generating single-instruction fetches and puts of the typed data. Without alignment constraints, on the other hand, the code might end up having to do two or more accesses spanning machine-word boundaries. Characters are a special case; theyre equally expensive from anywhere they live inside a single machine word. Thats why they dont have a preferred alignment.

casting to an odd pointer will slow down code and could work. Other architectures will take the word which the pointer points to and therefore the problem occurs above.

2.1.7 Javolution

2.1.8 Performance testing

2.2 Thread affinity

Thread.currentThread().getId(); just gets id of thread relative to jvm.

It keeps a process limited to certain a certain core or cores. Process will still be taken out of use and switched back in but without the problem of moving cache between cores.

Normally as a thread gets a time slice (a period in which to use the core), it is granted whichever core [CPU] is determined to be most free by the operating system's scheduler. Yes, this is in contrast to the popular fallacy that the single thread would stay on a single core. This means that the actual thread(s) of an application might get swapped around to non-overclocked cores, and even underclocked cores in some cases. As you can see, changing the affinity and forcing a single-threaded CPU to stay on a single CPU makes a big difference in such scenarios. The scaling up of a core does not happen instantly, not by a long shot in CPU time.

Therefore, for primarily single (or limited) thread applications, it is sometimes best to set the CPU affinity to a specific core, or subset of cores. This will allow the 'Turbo' processor frequency scaling to kick in and be sustained (instead of skipping around to various cores that may not be scaled up, and could even be scaled down).

core thrashing - ust by the name, you know this is a bad thing. You lose performance when a thread is swapped to a different core, due to the CPU cache being 'lost' each time. In general, the *least* switching of cores the better. One would hope the OS would try to avoid this, but it doesn't seem to at all in quick tests under Windows 7. Therefore, it is recommended you manually adjust the CPU affinity of certain applications to achieve better performance.

Another important issue is avoiding placing a load on a HyperThreaded (non-physical) core. These cores offer a small fraction of the performance of a real core. The Windows scheduler is aware of this and will swap to them only if needed. As of mid Jan 2012 the Windows 7 and Windows 2008 R2 schedulers have a hotfix for AMD Bulldozer CPUs that see them as HyperThreaded, cutting them down from 8 physical cores to 4 physical cores, 8 logical cores. This is for two reasons: The AMD Bulldozer platform uses pairs of cores called Bulldozer Modules. Each pair shares some computation units, such as an L2 cache and FPU. To spread out the load and prevent too much load being placed on two cores that have shared computational units, the Windows patch was released, boosting performance in lightly threaded scenarios.

Processor affinity takes advantage of the fact that some remnants of a process that was run on a given processor may remain in that processor's memory state (for example, data in the CPU cache) after another process is run on that CPU. Scheduling that process to execute on the same processor could result in an efficient use of process by reducing performance-degrading situations such as cache misses. A practical example of processor affinity is executing multiple instances of a non-threaded application, such as some graphics-rendering software.

```
1 cpu_set_t cpuset; \\ structure used to manage cpu affinity settings
2 pthread_t thread = pthread_self();  \\ get own system wide thread id
3 CPU_ZERO(&cpuset); \\ zero cpuset structure
4 CPU_SET(5, &cpuset); \\ set cpuset structure use with the 6th core (0-5)
```

```
5 int res = pthread_setaffinity_np(thread, sizeof(cpu_set_t), &cpuset); \\ set
      affinity of thread
6 \\ error handling here of res
7 res = pthread_getaffinity_np(thread, sizeof(cpu_set_t), &cpuset); \\ check
      affinity of thread
8 \\ error handling here of res
```

Code 5: Example of setting thread affinity

ref this

In Linux, Java thread uses the native thread(i.e, thread provided by Linux). This means the JVM creates a new native thread when the Java code creates a new Java thread. So, the Java threads can be organised in any way the native threads can be organised.

A native thread can be bound to a core through the pthread_setaffinity_np() function. So, a Java thread can be bound to a core. If Java standard library does not provide a function to do so, then this function need to be provided through JNI.

In Linux, multi-threading is same as parallel threading. Linux kernel distribute threads among processors to balance the cpu load. However the individual threads can be bound with any core as wished. So, in Linux Java multi-threading is same as parallel threading.

2.3 Endianness

This describes the order in which bytes of data types are stored in memory relative to pointers. In big-endian systems, the most significant bytes are stored at the pointer with every successive data bytes stored in successive memory locations. Conversely, in little-endian systems, the least significant byte is stored at the pointer. There is no advantage or disadvantage to either endian types, its simply a matter on convention for certain systems. Figure 1 shows how different endian systems store the value of the hexadecimal value *0x0A0B0C0D*.

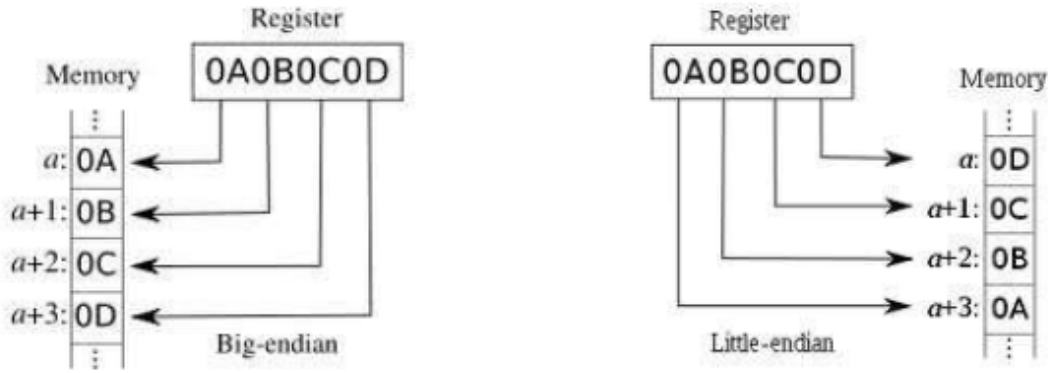

Figure 1: Difference between little and big endian

This becomes a problem when using the Java Native Interface with native code code on Intel architectures since the Java Virtual Machine uses big-endian format while Intel uses little-endian

format. Normally the JNI environment would handle this byte ordering conversion but since the JNI has been proven to be too slow to meet the requirement, this needs to be handled by the application.

ref this

2.4 Data type conversion

Between different languages the same data type can be represented by varying lengths in bytes and whether unsigned or signed (in the case of numerical values). It can also be the case that data type lengths in native languages differ depending on the architecture and whether it is 32-bit or 64-bit.

This becomes more of an issue when sharing data between C and Java. Java always uses a signed integer representation with the most significant bit representing whether the number is negative or positive. C uses both signed and unsigned representation depending on the requirements with varying byte length. Again, any conversion between the differences is normally handled via the JNI, but since the implementation aims to bypass the JNI as much as possible, data type conversion will have to be handled.

Lets take the integer representation as an example. In C, and integer can vary between 2, 4 and 8 bytes in length depending on the architecture and compiler. To solve this, standard integer types are used (e.g. uint8_t & int32_t) which guarantee that the representation is at least the length of the type definition stated. In Java, an integer is guaranteed to be 4 bytes long regardless of the system. However, DPDK uses unsigned integers while Java uses signed integers. This requires conversion between the unsigned and signed, but since unsigned has a higher upper bound on the value it can store due to the extra bit (MSB) there can be an overflow error when converting to Java. This requires that Java uses a long (8 byte) representation to hold the C 4 byte unsigned integer.

Conversion from Java to C then could then result in an underflow error if a value which can be represented by a Java long can't be represented by a C integer. This means bound checking is required on the Java side for any number conversions.

write java code to check and make own exception

2.5 Protocol undertaking

just IP (4 and) supporting udp and tcp (i think), no ARP etc

get a table of this