1 Is Java really slow?

Before any implementation or specific design considerations can be 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

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.

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 returne 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 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

1.4 Performance Testing Techniques

Benchmarking is a process of testing hardware, individual components or full end to end systems to determine the performance of the application or hardware . Generally, benchmarking should be repeatable under numerous iterations without only minor variations in performance results. This is firstly to allow minor changes to be made to the application/component with re-runs of the benchmark showing the performance changes. Secondly, it allows accurate comparisons to be drawn between similar software or hardware with different implementations in order to derive a better product.

Examples of hardware comparisons

better word for derive

why I need benchmarking

1.4.1 Programming Languages

It is well known that different programming languages can provide a radical change in execution for a given program. However, direct comparisons can't truly be trusted as certain languages are suited for for specific tasks and finding a benchmarking program to incorporate this is problematic. Other factors can be introduced when deciding on the optimisation level and the compiler of JIT used.

Numerous attempts have been made to compare languages, most noticeably the 'Benchmark Game' and Google.

show example of optimising java and how different it looks, also different depending on architecture

better phrasing off that sentence needed

More here

get paper

1.4.1.1 Loop Recognition

Google inducted their own experiment on this problem, testing only C++, Java, Scala and Go on the loop recognition algorithm . Implementations made use of standard looping constructs and memory allocation schemes without the use of non-orthodox optimisation techniques. Selected results of this are shown below:

Benchmark	Time [sec]	Facter
c++	23	1.0x
Java 64-bit	134	5.8x
Java 32-bit	290	12.8x
Java 32-bit GC	106	4.6x
Scala	82	3.6x
Go 6g	161	7.0x

Table 1: Results from Loop Recognition benchmarking

explain difference with GC etc

1.4.1.2 Benchmark Game

The Benchmark Game is an online community which aims to find the best programming language by using multiple benchmarking algorithms running on different architecture configurations to determine the outcome. Again, even this community regard the best benchmark application to be your application. A few selected results are shown below between Java and C (those used in this report) for a few different benchmarks.

get data

1.4.1.3 Using Economics

About economics paper

1.4.1.4 Which is better?

1.4.2 Intra-Language Techniques

1.4.3 Applications