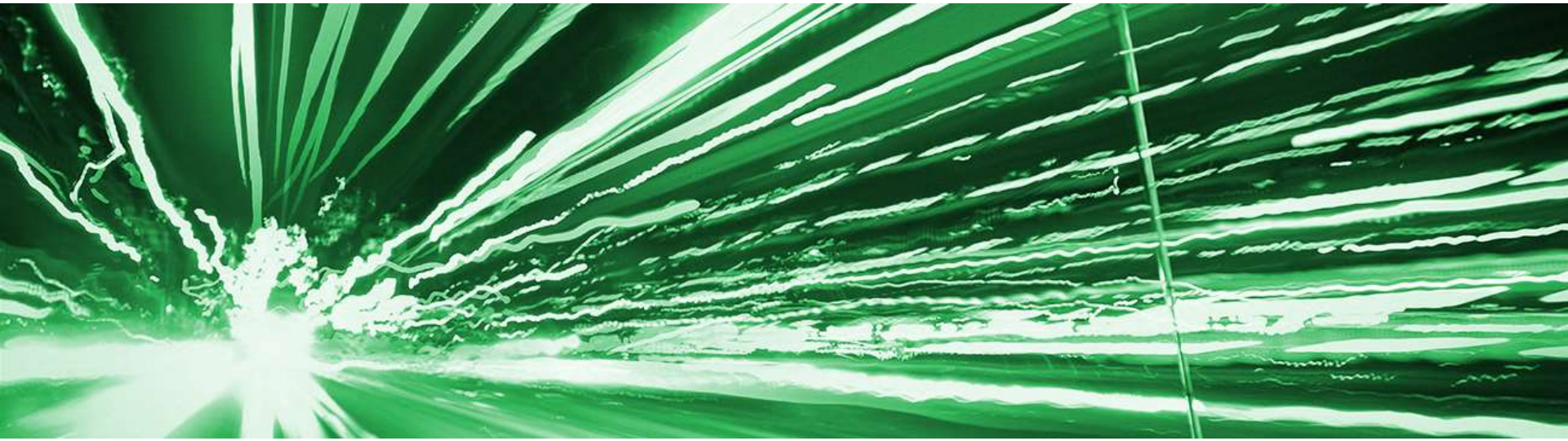
Chronicle



# Low Latency in Java 8

Peter Lawrey
CEO of Higher Frequency Trading
J On The Beach 2016

Chronicle

# Peter Lawrey

Java Developer/Consultant for investment banks and hedge funds for 8 years, 23 years in IT.

Most answers for Java and JVM on stackoverflow.com

Founder of the Performance Java User's Group.

Architect of Chronicle Software

Java Champion

Chronicle

# Chronicle Software

Help companies migrate to high performance Java code.

Sponsor open source projects https://github.com/OpenHFT

Licensed solutions Chronicle-FIX, Chronicle-Enterprise and Chronicle-Queue-Enterprise

Offer one week proof of concept workshops, advanced Java training, consulting and bespoke development.

# Chronicle

**FIX** – Micro seconds customisable FIX Engine

**Enterprise** – Monitoring, Traffic Shaping, Security

**Queue-Enterprise** – Confirmed Replication Distributed Queue

**Journal** – Custom Data Store, Key-Queue

**Engine** – Customisable Data Fabric, Reactive Live Queries

**Queue** – Persist every event

**Map** – Persisted Key-Value

**Wire** – YAML, Binary YAML, JSON, CSV, Raw data

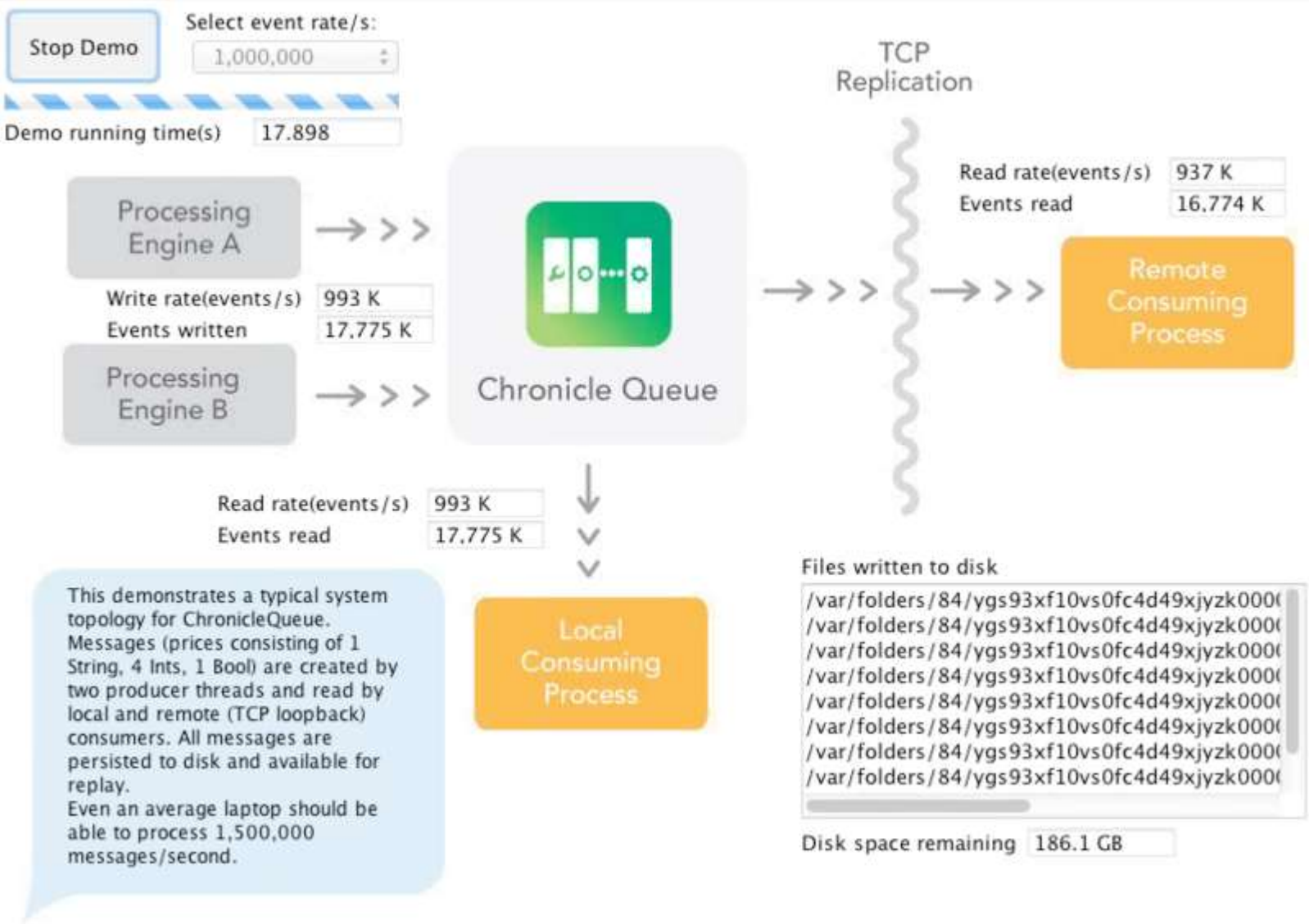
**Network** – Remote access

**Bytes** – 64-bit off heap native + memory mapped files

**Threads** – Low latency

**Core** – Low level access to OS and JVM

Chronicle


Stop Demo
Select event rate/s:
1,000,000
Demo running time(s) 17.898
TCP Replication
Read rate(events/s) 937 K
Events read 16,774 K
Processing Engine A
Write rate(events/s) 993 K
Events written 17,775 K
Processing Engine B
Chronicle Queue
Remote Consuming Process
Read rate(events/s) 993 K
Events read 17,775 K
Local Consuming Process
This demonstrates a typical system topology for ChronicleQueue.
Messages (prices consisting of 1 String, 4 Ints, 1 Bool) are created by two producer threads and read by local and remote (TCP loopback) consumers. All messages are persisted to disk and available for replay.
Even an average laptop should be able to process 1,500,000 messages/second.
Files written to disk
/var/folders/84/ygs93xf10vs0fc4d49xjyzk000(
/var/folders/84/ygs93xf10vs0fc4d49xjyzk000(
/var/folders/84/ygs93xf10vs0fc4d49xjyzk000(
/var/folders/84/ygs93xf10vs0fc4d49xjyzk000(
/var/folders/84/ygs93xf10vs0fc4d49xjyzk000(
/var/folders/84/ygs93xf10vs0fc4d49xjyzk000(
/var/folders/84/ygs93xf10vs0fc4d49xjyzk000(
Disk space remaining 186.1 GB

Chronicle

# Agenda

- What is Low Latency?
- What is an ultra low GC rate?
  - Eden space and Compressed OOPs.
- How does Java 8 help?
  - Non-capturing lambdas
  - Serializing Lambdas
  - Escape Analysis
- Optimise for Latency distribution not Throughput.
  - Flow control and avoiding co-ordinated omission.
- Low latency code using Java 8.

Chronicle

# What is low latency?

The term “low latency” can applied to a wide range of situations.

A broad definition might be;

Low latency means you have a view on how much the response time of a system costs your business.

In this talk I will assume;

Low latency means you care about latencies you can only measure as even the worst latencies are too fast to see.

Chronicle

# Even latencies you can't see add up

| Data passing | Latency | Light over a fibre | Throughput on at a time |
|---|---|---|---|
| Method call | Inlined: 0<br>Real call: 50 ns. | 10 meters | 20,000,000/sec |
| Shared memory | 200 ns | 40 meters | 5,000,000/sec |
| SYSV Shared memory | 2 µs | 400 meters | 500,000/sec |
| Low latency network | 8 µs | 1.6 km | 125,000/sec |
| Typical LAN network | 30 µs | 6 km | 30,000/sec |
| Typical data grid system | 800 µs | 160 km | 1,250/sec |
| 60 Hz power flickers | 8 ms | 1600 km | 120/sec |
| 4G request latency in UK | 55 ms | 11,000 km | 18/sec |

Chronicle

# Why consistent performance matters.

When you have high delays, even rarely, this two consequences

- Customers tend to remember the worst service they ever got. You tend to lose the most money when are unable to react to events the longest.
- In a busy system, a delayed in processing an event/message/request has a knock on effect many subsequent requests.  A single delay can impact hundreds or even many thousands of requests.

Chronicle

# What is an ultra low GC?

In a generational collector you have multiple generations of objects.

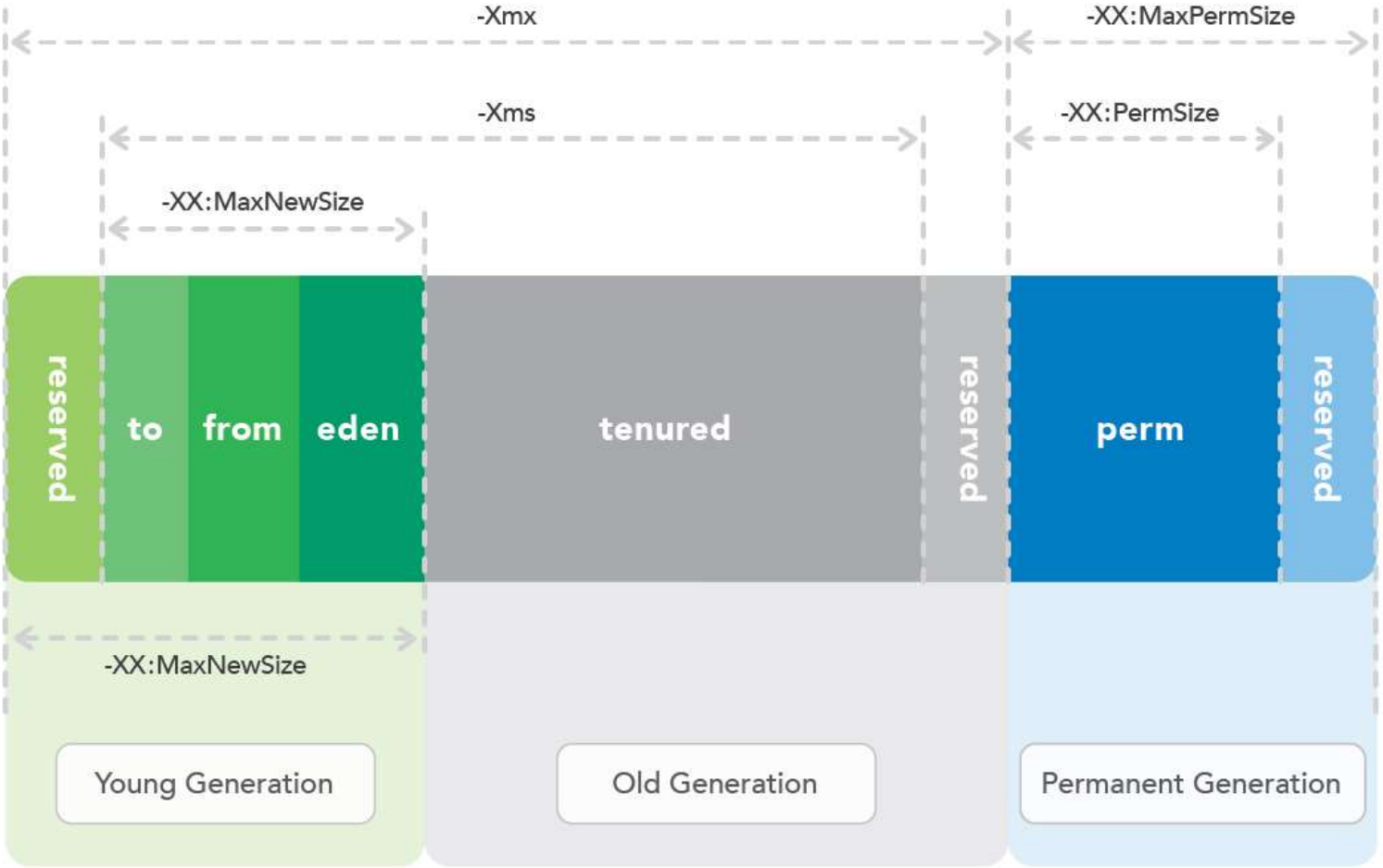
Small to medium sized objects are created in the Eden space.

When your Eden space fills up you trigger a Minor Collection.

So how big can your Eden space be and still have Compressed OOPS?

Chronicle

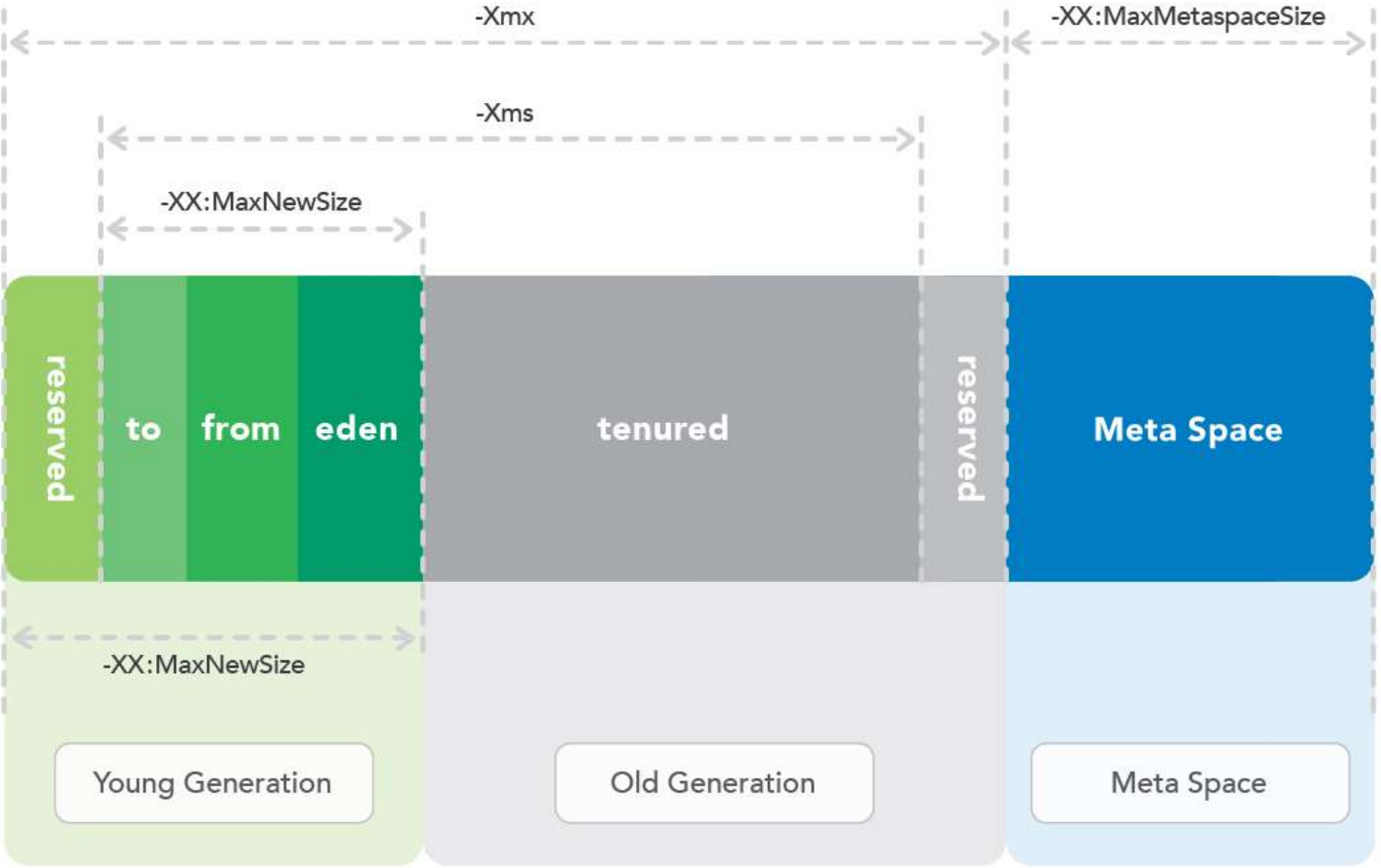
# Where does the Eden space fit in




*Java 7 memory layout.*

Chronicle

# Where does the Eden space fit in




*Java 8 memory layout.*

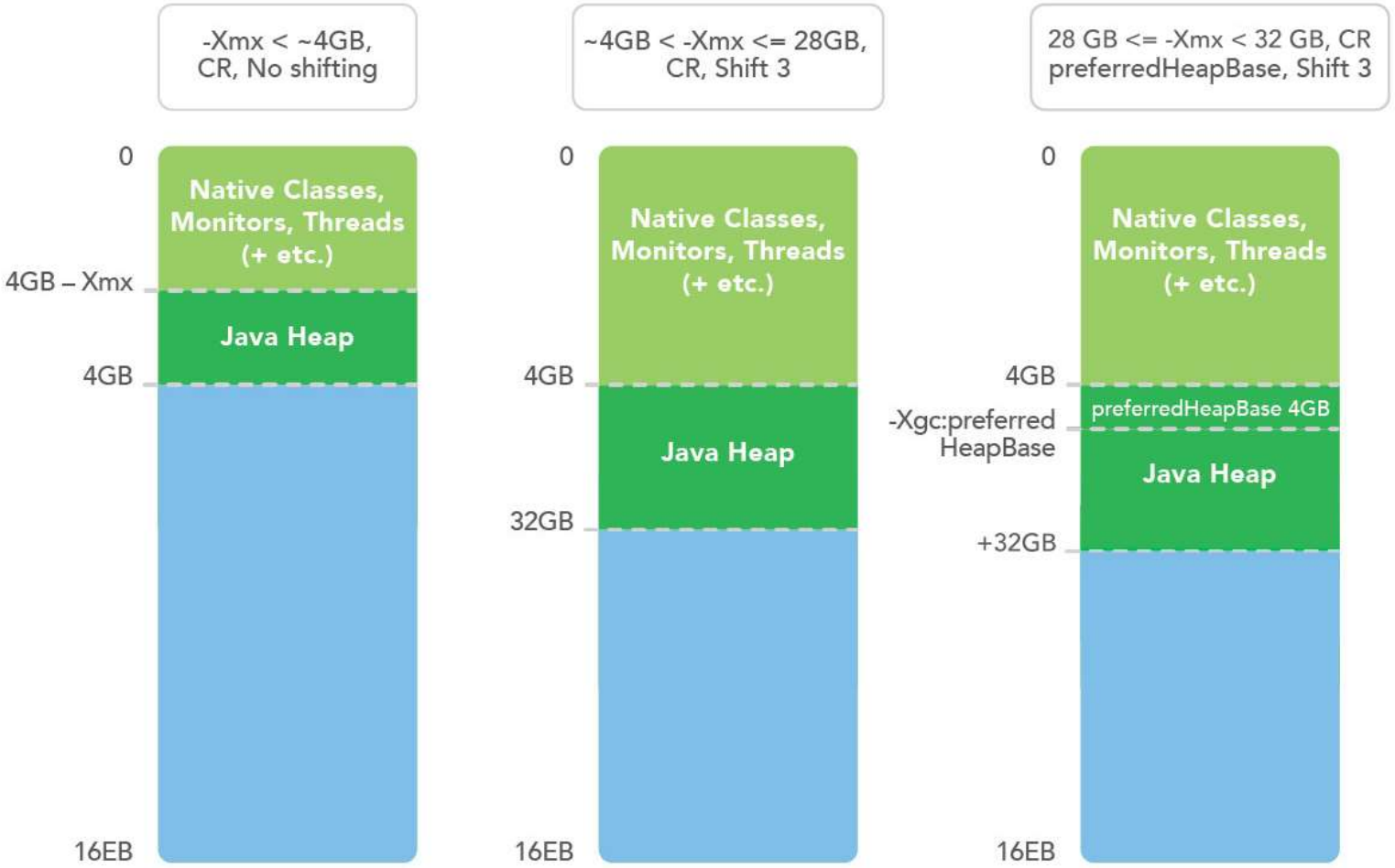
# What are Compressed OOPS?

64-bit applications use 64-bit pointers. This take up more space than 32-bit pointers. For real applications this can mean up to 30% of memory needed.

However, the 64-bit JVM can address the heap using 32-bit *references.* This saves memory and improves the efficiency of the cache as you can keep more objects in cache.

Chronicle

# What are Compressed OOPS?



Chronicle

# What are Compressed OOPS?

*In Java 8*; If the heap is between 32 GB and 64 GB …

```
-XX:+UseCompressedOops
-XX:ObjectAlignmentInBytes=16
```

The JVM can address anywhere on the heap by multiplying by 16 and adding a relative offset.

You can increase the object alignment to 32 manually but at this point 64-bit references uses less memory.

Chronicle

# Ultra low GC

Let say you have an Eden size of 48 GB.

You will only get a garbage collection once the Eden space fills, ie you have used 48 GB.

If you need to run for 24 hours without a GC, you can still produce 2 GB of garbage, and GC once per day.

This is a rate of 500KB/s,

# What is an ultra low GC?

Another reason to use ultra low garbage rates is your caches are not being filled with garbage and they work much more efficiently and consistently.

If you have a web server which is producing 300 MB/s of garbage, this means less than 5% of the time you will be pausing for a GC.

However, it does mean that you could be filling a 32 KB L1 CPU in around 0.1 milli-seconds. Your L2 cache fills with garbage every milli-second.

Chronicle

# So without a GC pause, no more pauses?

Actually, a high percentage of pauses are not from the GC. The biggest ones are, but take away GC pauses and still see;

- IO delays. These can be larger than GC pauses.
  - Network delays.
  - Waiting for databases.
  - Disk reads / writes.
- OS interrupts.
  - It is not uncommon for your OS to stop your process for 5 ms or more.
- Lock contention pauses.

# How does Java 8 help?

The biggest improvement in Java 8 are;

- Lambdas with no captured values and lambdas with reduced capture of variables.
  - More efficient than anonymous inner classes.
- Escape Analysis to unpack objects onto the stack.
  - Short lived objects placed on the stack don’t create garbage.

# How do Lambdas help?

Lambdas are like anonymous inner classes, however they are assigned to static variables if they don't capture anything.

```
public static Runnable helloWorld() {
    return () -> System.out.println("Hello World");
}

public static Consumer<String> printMe() {
    // may create a new object each time = Garbage.
    return System.out::println;
}

public static Consumer<String> printMe2() {
    return x -> System.out.println(x);
}
```

Chronicle

# How does Java 8 help? Lambdas

When you call *new* on an anonymous inner classes, a new object is always created. Non capturing lambdas can be cached.

```
Runnable r1 = helloWorld();
Runnable r2 = helloWorld();
System.out.println(r1 == r2); // prints true

Consumer<String> c1 = printMe();
Consumer<String> c2 = printMe();
System.out.println(c1 == c2); // prints false

Consumer<String> c3 = printMe2();
Consumer<String> c4 = printMe2();
System.out.println(c3 == c4); // prints true
```

# Serialization

Lambdas capture less scope.  This means it doesn't capture *this* unless it has to, but it can capture things you don't expect.

Lambdas capture less scope.  If you use *this.a* the value of *a* is copied.

Note: if you use the :: notation, it will capture the left operand if it is a variable.

Chronicle

```
interface SerializableConsumer<T> extends Consumer<T>, Serializable {
}

// throws java.io.NotSerializableException: java.io.PrintStream
public SerializableConsumer<String> printMe() {
    return System.out::println;
}

public SerializableConsumer<String> printMe2() {
    return x -> System.out.println(x);
}

public SerializableConsumer<String> printMe3() {
    // throws java.io.NotSerializableException: A
    return new SerializableConsumer<String>() {
        @Override
        public void accept(String s) {
            System.out.println(s);
        }
    };
}
```

Chronicle

# Why Serialize a Lambda?

Lambdas are designed to reduce boiler plate, and when you have a distributed system, they can be a powerful addition.

```
public static long incrby(MapView<String, Long> map, String key, long toAdd) {
    return map.syncUpdateKey(key, v -> v + toAdd, v -> v);
}
```

The two lambdas are serialized on the client to be executed on the server.

This example is from the RedisEmulator in Chronicle-Engine.

Chronicle

# Why Serialize a Lambda?

The lambda m-> { is serialized and executed on the server.

```
public static Set<String> keys(MapView<String, ?> map, String pattern) {
    return map.applyTo(m -> {
        Pattern compile = Pattern.compile(pattern);
        return m.keySet().stream()
                .filter(k -> compile.matcher(k).matches())
                .collect(Collectors.toSet());
    });
}
```

Chronicle

# Why Serialize a Lambda?

The filter/map lambdas are serialized.

The subscribe lambda is executed asynchronously on the client.

```
// print userId which have a usageCounter > 10
// each time it is incremented (asynchronously)

userMap.entrySet().query()
        .filter(e -> e.getValue().usageCounter > 10)
        .map(e -> e.getKey())
        .subscribe(System.out::println);
```

# How does Java 8 help? Escape Analysis

Escape Analysis can

- Determine an object doesn’t escape a method so it can be placed on the stack.
- Determine an object doesn’t escape a method so it doesn’t need to be synchronized.

# How does Java 8 help? Escape Analysis

Escape Analysis works with inlining. After inlining, the JIT can see all the places an object is used. If it doesn't escape the method it doesn't need to be created and can be unpacked on the stack. This works for class objects, but not arrays currently.

After "unpacking" on to the stack the object might be optimised away. E.g. say all the fields are set using local variables anyway.

# How does Java 8 help? Escape Analysis

As of Java 8 update 60, the JITed code generated is still not as efficient as code written to no need these optimisation, however the JIT is getting closer to optimal.

Chronicle

# How does Java 8 help? Escape Analysis

To parameters which control in-lining and the maximum method size for performing escape analysis are

```
-XX:MaxBCEAEstimateSize=150  -XX:FreqInlineSize=325
```

For our software I favour

```
-XX:MaxBCEAEstimateSize=450  -XX:FreqInlineSize=425
```

Chronicle

# Optimising for latency instead of throughput

Measuring Throughput is a great way to hide bad service, or poor latencies.

Your users however tend to be impacted by the poor services/latencies.

Chronicle

# Optimising for latency instead of throughput

Average latency is largely an inverse of your throughput and no better. Using standard deviation for average latency is mis-leading at best as the distribution for latencies are not a normal distribution or anything like it.

From Little's Law

Average latency = concurrency / throughput.

Chronicle

# How to hide bad performance with average latency

Say you have a service which responds to 2 requests every milli-seconds, but once an hour it stops for 6 minutes.

Without the 6 minute pause, the average latency would **0.5** ms.

With the 6 minute latency, the number of tasks performed in an hour drop from 7.2 million to 6 million and the average latency is **0.6** ms.

*Conclusion*: pausing for 6 minutes each hour doesn't make much difference to our metric.

Chronicle

# How to find solvable problems with latency distributions

Run a test with many time at a given rate.

Time how long the task takes from when the test *should have started*, not when you actually started.

Sort these timings and look at the 99%, 99.9%, 99.99% and worst numbers in your results.

The 99% (or worst 1 in 100) will be much higher than your average latency and is more like to explain why your users see bad performance sometimes.

Ideally you should reduce your 99.9% and even your worst latency.

Chronicle

# Is it fair to time from when the test should have started?

Without having a view on when your tests should have started you don't consider that if the system stalls, it could impact tens or even thousands of tasks. i.e. you can't know the impact of a long delay.

Unfortunately a lot of tools optimistically assume only 1 tasks ways delayed, but this is unrealistic.

Chronicle

# Co-ordinated omission

Term coined by Gill Tene, CTO Azul Systems.

Co-ordinated omission occurs when you have a benchmark which accepts flow control from the solution being tested.

Flow control allows the system being tested to stop the benchmark when it is running slow.  Most older benchmark tools allow this!!

# What is flow control?

Most producer/consumer systems use some for of flow control.

Flow control allows the consumer to stop the producer to prevent the system from becoming overloaded and cause a failure of the system.

TCP/IP uses flow control for example.

UDP doesn’t have flow control and if the consumer can’t keep up, the messages are lost.

# What is flow control?

Chronicle Queue however uses an open ended persisted queue. This avoid the need for flow control as the consumer can be any amount behind the producer (to the limits of your disk capacity)

A system without flow control is easier to reproduce for testing debugging and performance tuning purposes.

You can test the producer or consumer in isolation as they don’t interact with one another.

# Why is flow control bad for performance measures?

Flow control helps deal with period when the consumer can't keep up. It does so by slowing down or stopping the producer.

For performance testing this is like creating a blind spot for the producer which is the load generator.

The load generator measures all the times the consumer can keep up, but significantly underrates when the consumer can't keep up.

Chronicle

# Our example without co-ordinated omission

Say you have a service which responds to 2 requests every milli-seconds, but once an hour it stops for 6 minutes.

In the 6 minutes when the process stopped, how many tasks were delayed.  The optimistic answer is 1, but the pessimistic answer is 2 every milli-second or 720,000.

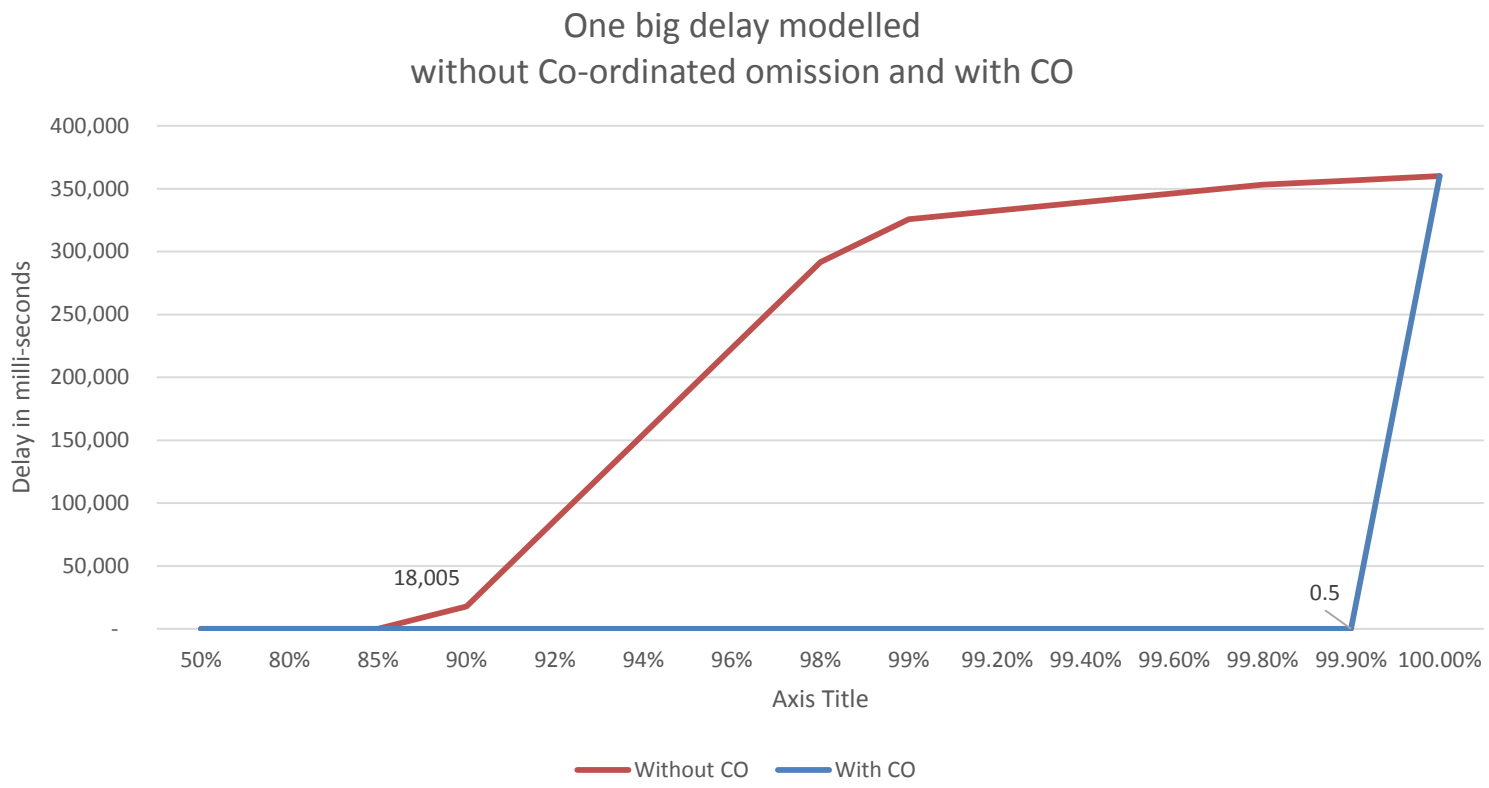
# Our example without co-ordinated omission

This is why the expect rate matters, and unless you see an expected throughput used in tested, there is a good chance co-ordinated omission occurred.

Lets consider that the target throughput was 100 per second. In the 6 minutes at least 6 * 60 * 100 requests were delayed. Some 6 minutes, some 5 minutes, … 1 minute. However as the requests are being performed, more tests are being added. In the 18 seconds it takes process the waiting requests, about 1800 are added but soon the queue is empty.

Chronicle

# Co-ordinated omission



Chronicle

# Optimising for latency instead of throughput

Having enough throughput is only the start.  You also ned to look at the consistency of your service.

The next step is to look at your 99 percentile latencies (worst 1 in 100).  After that you can look at your 99.9% tile, 99.99% tile and your worst latencies tested.

Chronicle

# A low latency API which uses Lambdas

Chronicle Wire is a single API which supports multiple formats. You decide what data you want to read/write and independently you can chose the format. E.g. YAML, JSON, Binary YAML, XML.

Using lambdas helped to simplify the API.

Chronicle

# A low latency API which uses Lambdas

| Wire Format | Bytes | 99.9 %tile | 99.99 %tile | 99.999 %tile | worst |
|---|---|---|---|---|---|
| JSONWire | 100* | 3.11 | 5.56 | 10.6 | 36.9 |
| Jackson | 100 | 4.95 | 8.3 | 1,400 | 1,500 |
| Jackson + Chronicle-Bytes | 100* | 2.87 | 10.1 | 1,300 | 1,400 |
| BSON | 96 | 19.8 | 1,430 | 1,400 | 1,600 |
| BSON + Chronicle-Bytes | 96* | 7.47 | 15.1 | 1,400 | 11,600 |
| BOON Json | 100 | 20.7 | 32.5 | 11,000 | 69,000 |

Timings are in micro-seconds with JMH.

```
"price":1234,"longInt":1234567890,"smallInt":123,"flag":true,"text":"Hello World!","side":"Sell"
```

* Data was read/written to native memory.

Chronicle

# A resizable buffer and a Wire format

```
// Bytes which wraps a ByteBuffer which is resized as needed.
Bytes<ByteBuffer> bytes = Bytes.elasticByteBuffer();
// YAML based wire format
Wire wire = new TextWire(bytes);

// or a binary YAML based wire format
Bytes<ByteBuffer> bytes2 = Bytes.elasticByteBuffer();
Wire wire2 = new BinaryWire(bytes2);

// or just data, no meta data.
Bytes<ByteBuffer> bytes3 = Bytes.elasticByteBuffer();
Wire wire3 = new RawWire(bytes3);
```

```
message: Hello World number: 1234567890 code: SECONDS price: 10.5
```

# Low latency API using Lambdas (Wire)

## To write a message

```
wire.write(() -> "message").text(message)
        .write(() -> "number").int64(number)
        .write(() -> "timeUnit").asEnum(timeUnit)
        .write(() -> "price").float64(price);
```

## To read a message

```
wire.read(() -> "message").text(this, (o, s) -> o.message = s)
        .read(() -> "number").int64(this, (o, i) -> o.number = i)
        .read(() -> "timeUnit").asEnum(TimeUnit.class, this, (o, e) -> o.timeUnit = e)
        .read(() -> "price").float64(this, (o, d) -> o.price = d);
```

```
message: Hello World number: 1234567890 code: SECONDS price: 10.5
```

# A resizable buffer and a Wire format

## In the YAML based TextWire

```
message: Hello World
number: 1234567890
code: SECONDS
price: 10.5
```

## Binary YAML Wire

```
00000000 C7 6D 65 73 73 61 67 65 EB 48 65 6C 6C 6F 20 57 ·message ·Hello W
00000010 6F 72 6C 64 C6 6E 75 6D 62 65 72 A3 D2 02 96 49 orld·num ber····I
00000020 C4 63 6F 64 65 E7 53 45 43 4F 4E 44 53 C5 70 72 ·code·SE CONDS·pr
00000030 69 63 65 90 00 00 28 41 ice···(A
```

```
message: Hello World number: 1234567890 code: SECONDS price: 10.5
```

# Lambdas and Junit tests

## To read the data

```
wire.read(() -> "message").text(this, (o, s) -> o.message = s)
      .read(() -> "number").int64(this, (o, i) -> o.number = i)
      .read(() -> "timeUnit").asEnum(TimeUnit.class, this, (o, e) -> o.timeUnit = e)
      .read(() -> "price").float64(this, (o, d) -> o.price = d);
```

## To check the data without a data structure

```
wire.read(() -> "message").text("Hello World", Assert::assertEquals)
      .read(() -> "number").int64(1234567890L, Assert::assertEquals)
      .read(() -> "timeUnit").asEnum(TimeUnit.class, TimeUnit.SECONDS,Assert::assertEquals)
      .read(() -> "price").float64(10.5, (o, d) -> assertEquals(o, d, 0));
```

```
message: Hello World number: 1234567890 code: SECONDS price: 10.5
```

# Interchanging Enums and Lambdas

Enums and lambdas can both implement an interface.

Wherever you have used a non capturing lambda you can also use an enum.

```
enum Field implements WireKey {
    message, number, timeUnit, price;
}

@Override
public void writeMarshallable(WireOut wire) {
    wire.write(Field.message).text(message)
            .write(Field.number).int64(number)
            .write(Field.timeUnit).asEnum(timeUnit)
            .write(Field.price).float64(price);
}
```

```
message: Hello World number: 1234567890 code: SECONDS price: 10.5
```

# When to use Enums

Enums have a number of benefits.

- They are easier to debug.
- The serialize much more efficiently.
- Its easier to manage a class of pre-defined enums to implement your code, than lambdas which could be any where

Under https://github.com/OpenHFT/Chronicle-Engine search for MapFunction and MapUpdater

```
message: Hello World number: 1234567890 code: SECONDS price: 10.5
```



# When to use Lambdas

Lambdas have a number of benefits.

- They are simpler to write
- They support generics better
- They can capture values.

```
message: Hello World number: 1234567890 code: SECONDS price: 10.5
```

# Where can I try this out?

The source for these micro-benchmarks are test are available

https://github.com/OpenHFT/Chronicle-Wire

Chronicle Engine with live subscriptions

https://github.com/OpenHFT/Chronicle-Engine

Chronicle

# Q & A

Peter Lawrey

@PeterLawrey

http://chronicle.software

http://vanillajava.blogspot.com