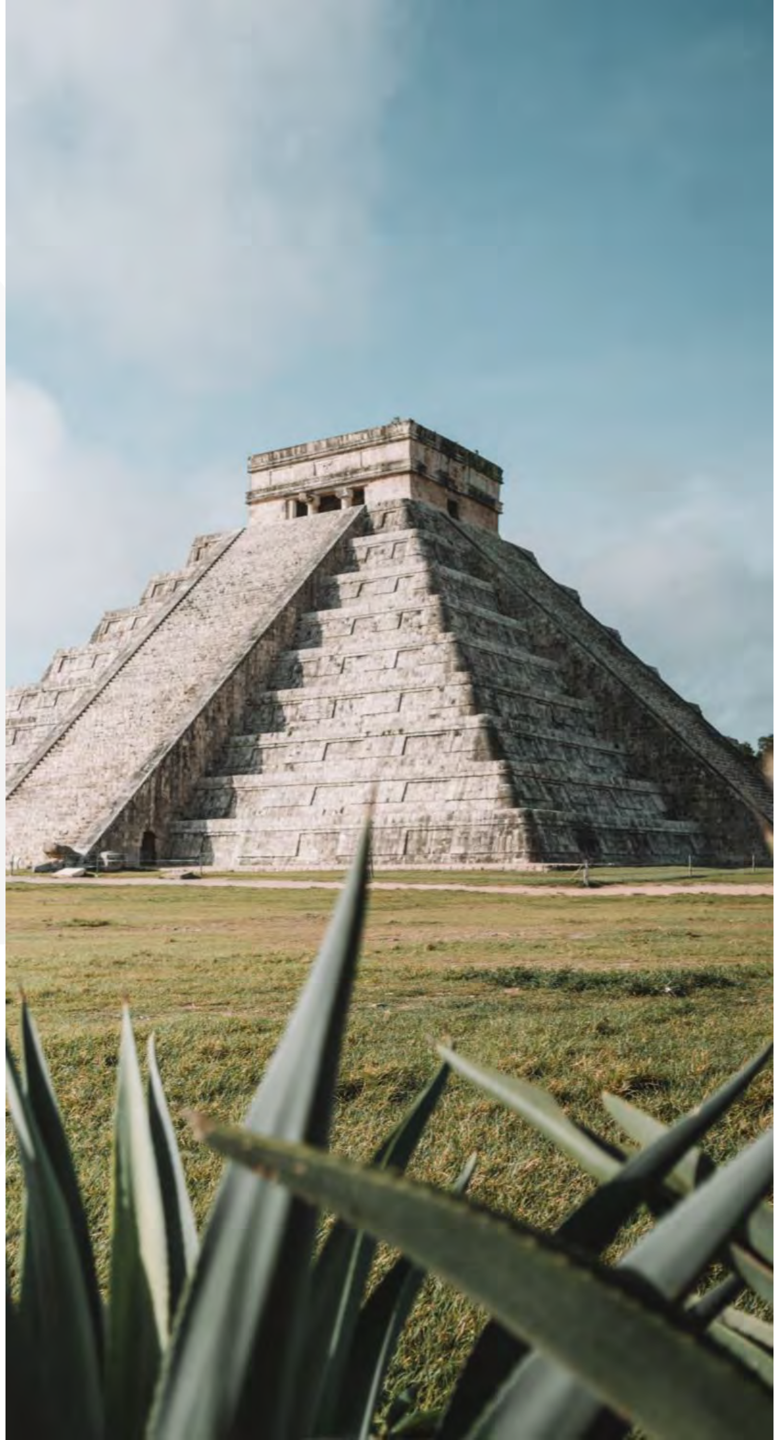
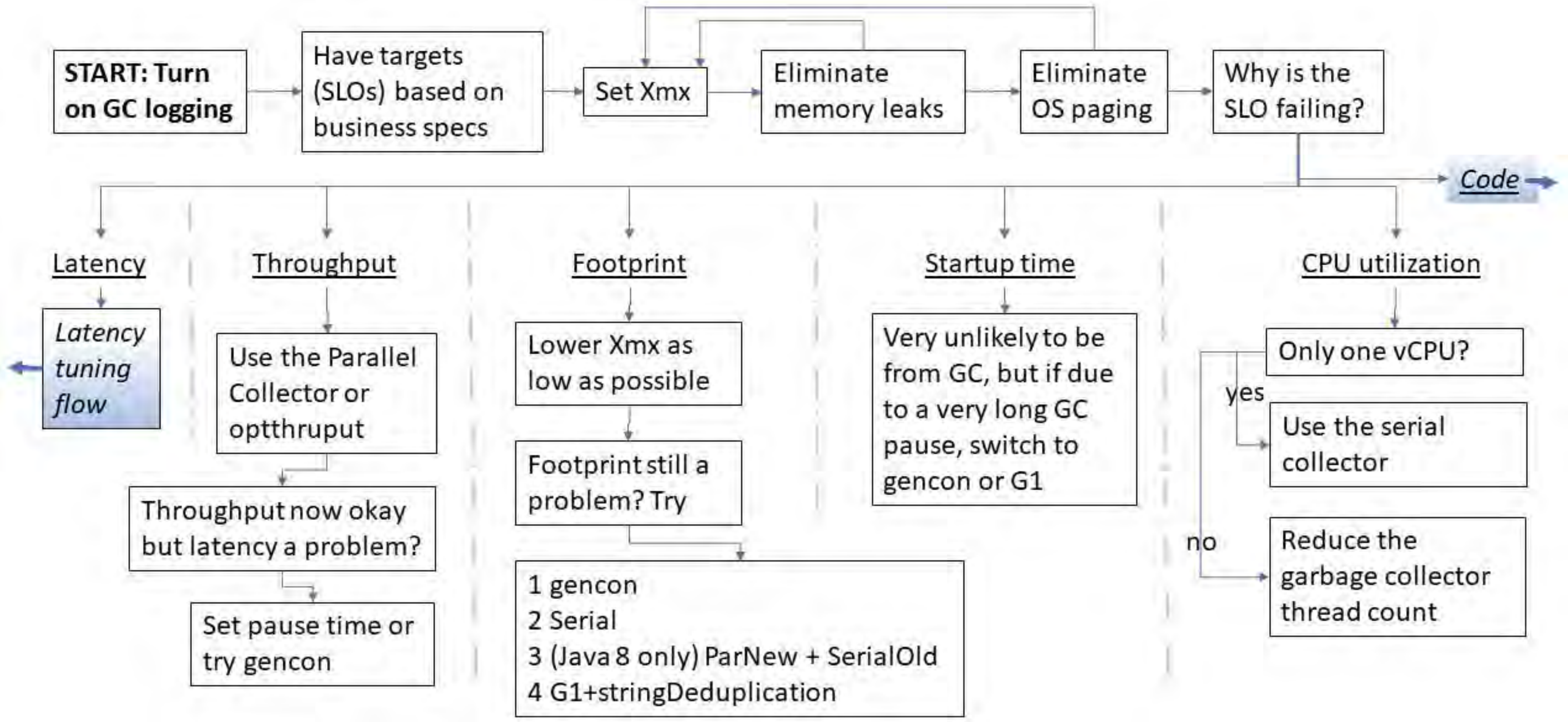
expedia group™

Modern Garbage Collection Tuning – Step by Step

Jack Shirazi

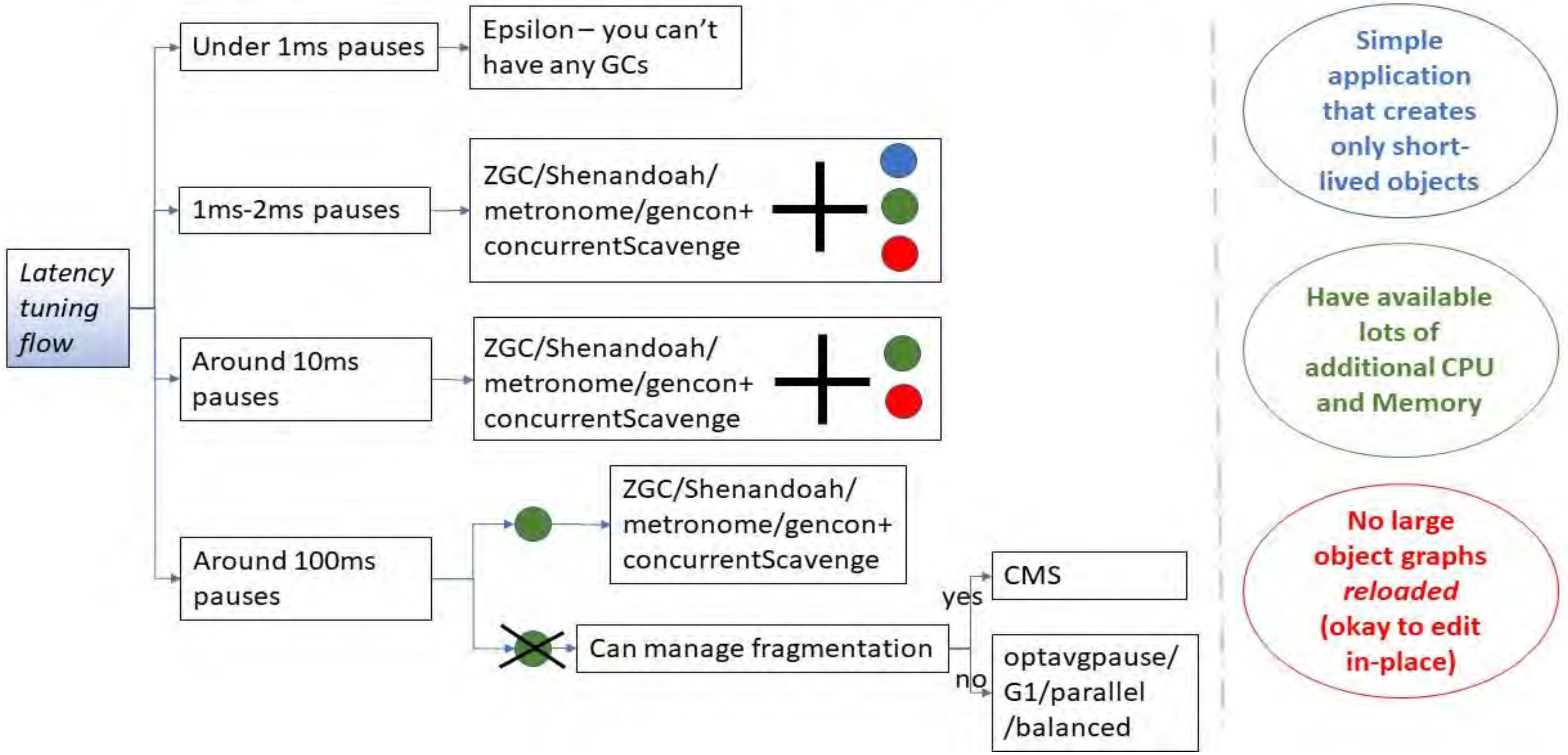


JVM 8-14 Garbage Collection Tuning Flow



START: Turn on GC logging
Have targets (SLOs) based on business specs
Set Xmx
Eliminate memory leaks
Eliminate OS paging
Why is the SLO failing?
Code
Latency
Latency tuning flow
Throughput
Use the Parallel Collector or optthruput
Throughput now okay but latency a problem?
Set pause time or try gencon
Footprint
Lower Xmx as low as possible
Footprint still a problem? Try
1 gencon
2 Serial
3 (Java 8 only) ParNew + SerialOld
4 G1+stringDeduplication
Startup time
Very unlikely to be from GC, but if due to a very long GC pause, switch to gencon or G1
CPU utilization
Only one vCPU?
yes
Use the serial collector
no
Reduce the garbage collector thread count

JVM 8-14 Garbage Collection **Latency** Tuning Flow



Latency tuning flow
Under 1ms pauses
Epsilon – you can't have any GCs
1ms-2ms pauses
ZGC/Shenandoah/ metronome/gencon+ concurrentScavenge
Around 10ms pauses
ZGC/Shenandoah/ metronome/gencon+ concurrentScavenge
Around 100ms pauses
ZGC/Shenandoah/ metronome/gencon+ concurrentScavenge
Can manage fragmentation
yes
CMS
no
optavgpause/ G1/parallel /balanced
Simple application that creates only short-lived objects
Have available lots of additional CPU and Memory
No large object graphs *reloaded* (okay to edit in-place)

Choices

- OpenJDK has more than 120 GC tuning flags, 12 different garbage collectors, multiple memory spaces, pools and insanely varied GC logs

Hello

Options options options

- More options are
  - Yay!!!

Hello

Options options options

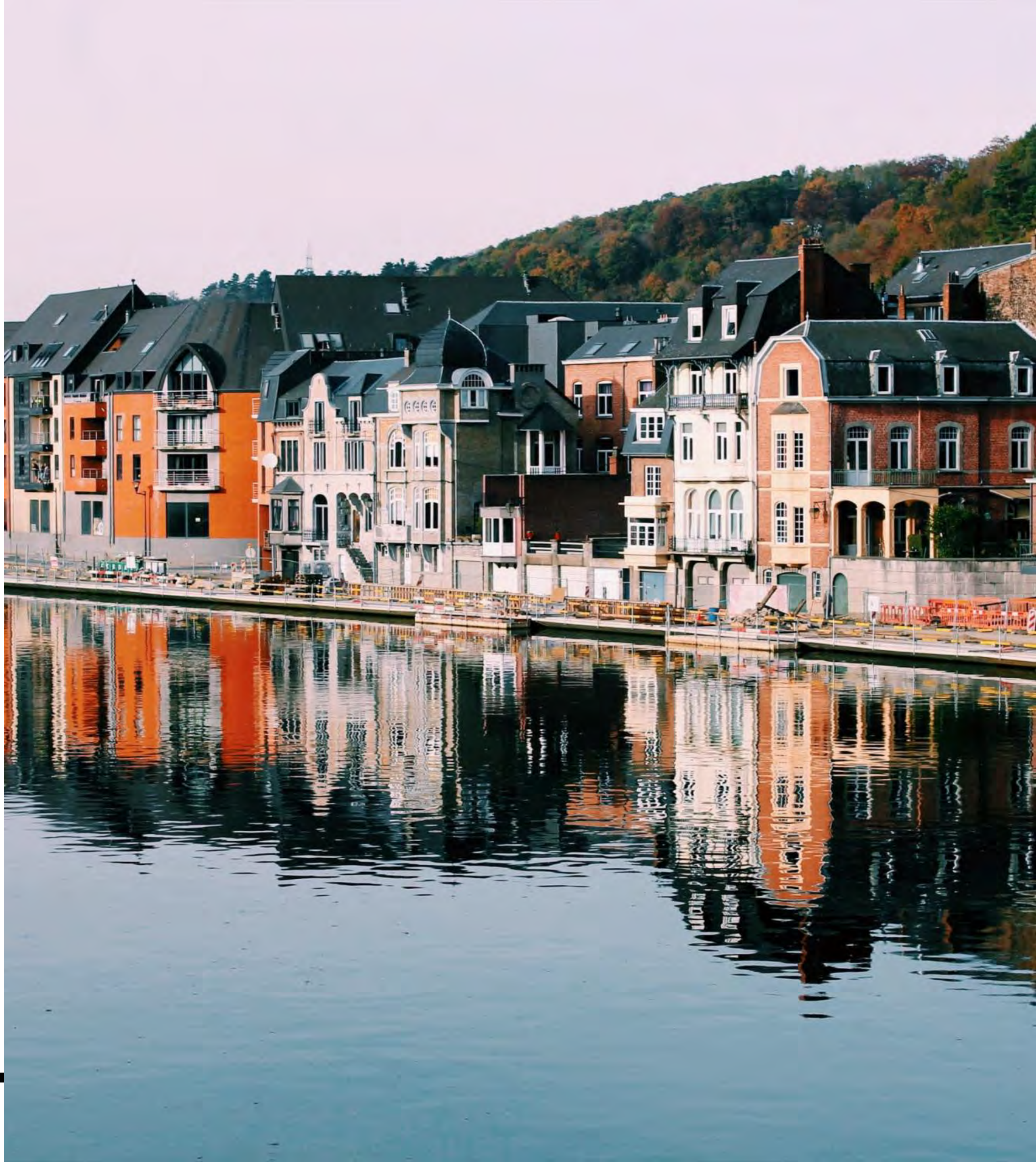
- More options are
  - Yay!!!
  - Shoot me now

Options options options

- More options are
  - Yay!!!
  - Shoot me now
- Don’t Panic

Before tuning

Some “Useful to Know” GC things



expedia group™

Jack Shirazi

Useful to know

The collector doesn't just collect garbage

- It also
  - decides on the memory spaces and layout
  - the size and layout of objects
  - how allocation happens
  - tracks objects over their life
  - adds in JIT compilation codes
  - re-layouts space and memory as needed to optimize (eg compaction)
- So choosing the garbage collector affects everything!

A lot of garbage collectors have 2 generations

- Usually called "young" and "old“
- Because they can use a quick but less accurate collector in the young generation (might leave garbage in memory)

- One tuning consequence: Try to have objects collected in the young generation

- Objects get "promoted" to the old generation where they are "tenured"

Useful to know

Collector words and phrases I

1/2

- Heap
- GC roots
- Mark-and-sweep
- Compaction
- Copying collector
- Concurrent vs parallel
- Concurrent vs stop-the-world

Useful to know

Collector words and phrases II

2/2

- Region
- Pause
- Safepoint and pause time
- Promotion and tenuring and tenuring age
- Finalizers & References
- Metaspace and permspace

Useful to know

OpenJDK Garbage Collectors

- For OpenJDK Java 8 (with backports), Java 11-15 there are 7 GC algorithms available with the *HotSpot* JVM and another 6 in the *OpenJ9* JVM - but one (Epsilon) is the same in each, so 12 in total. From Java 14, one of these (CMS) was removed.
- For all collectors the primary tuning option is the -Xmx maximum heap size. Usually more is better for low pause GCs. It's always worth trying out a different GC algorithm if your application is not meeting its SLOs and the cause of those SLO violations is significantly from GCs.

Useful to know

OpenJDK Garbage Collectors

No GC, Serial, Throughput

- Epsilon (-XX:+UseEpsilonGC/-Xgcpolicy:nogc) terminate rather than GC

- Serial (-XX:+UseSerialGC) targeted at 1 vCPU

- Parallel (-XX:+UseParallelGC) targeted at throughput
- Throughput (-Xgcpolicy:optthruput) targeted at throughput

OpenJDK Garbage Collectors

Targeted at pause time

- CMS (-XX:+UseConcMarkSweepGC) *(gone from JDK14+)*
- G1 (-XX:+UseG1GC)
- ZGC (-XX:+UseZGC)
- Shenandoah (-XX:+UseShenandoahGC)
- Balanced (-Xgcpolicy:balanced)
- Generational Concurrent (-Xgcpolicy:gencon)
- Metronome (-Xgcpolicy:metronome)
- Pause optimized (-Xgcpolicy:optavgpause)

The tuning flow

Overview – 4 possible tuning steps

- 1. Adjust heap size
- 2. Choose an appropriate collector
  - This talk is mainly here (+ a little on the next two sections)
- 3. Reduce the rate of object allocation and promotion
  - a) Adjust young generation heap size
  - b) Adjust tenuring threshold
  - c) Change code
- 4. Fine tune the GC algorithm

Turn on GC logging

- Negligible overhead
- -Xlog:gc*=info
- -Xlog:gc*=info,safepoint :file=<path>/logs/gc_%t.log: tags,time,uptime,level:filecount=10,filesize=50M

  - Java 8 Hotspot
  - -XX:+PrintGCDetails -XX:+PrintGCDateStamps -Xloggc:<path>/logs/gc_$(date +%Y_%m_%d-%H_%M).log -XX:+UseGCLogFileRotation -XX:NumberOfGCLogFiles=10 -XX:GCLogFileSize=50M -XX:+PrintGCApplicationStoppedTime

Have SLOs (targets)

- "time is money" is an expression
- But quite literal in this case
- Every millisecond tighter you specify will cost you in compute resources and development time and devops time
- Business SLOs, not GC ones!

The Far Side by Gary Larson Vintage 1984"BLANK INSIDE" Greeting Card with Envelope -"Einstein discovers that time IS actually money"

by Far Side by Gary Larson

Currently unavailable.

We don't know when or if this item will be back in stock.

amazonbasics Save on Quality Copy Papers Sh

Screenshot from https://www.amazon.com/Far-Side-Gary-Larson-discovers/dp/B07KGH621Q

Note thefarside.com is up too

The defaults

- Dictionary (noun)
  - "default"
    - **Meaning** not optimal
    - **(but sometimes good enough)**

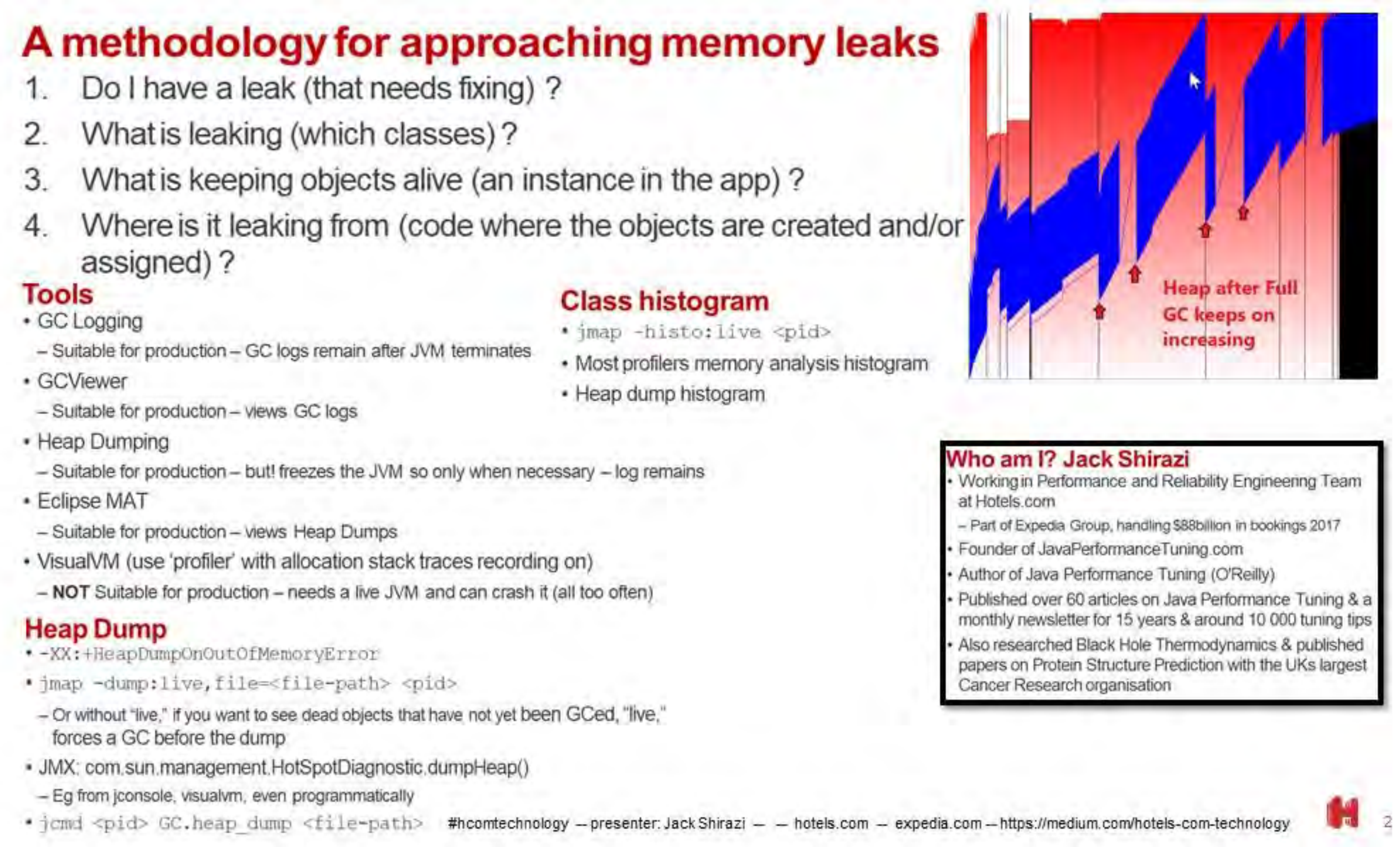
- Reasonable place to start

Xmx

- Start with Xmx set at 2x your live set
  - The live set is the stable size of the heap after GCs
- Then adjust as needed in response to SLOs
  - Lower for
    - Smaller footprint
    - More frequent but shorter GCs
  - Higher for
    - Better pause times when using concurrent algorithms

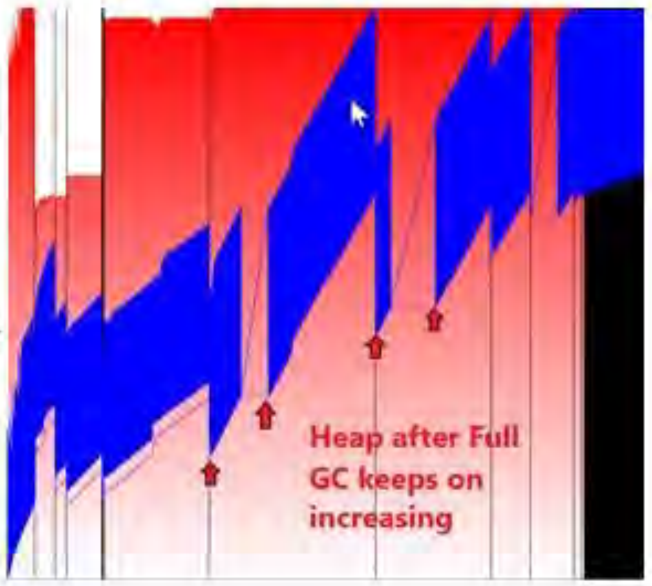
Eliminate Memory Leaks

- There is not much point in tuning the GC with a memory leak, no matter what you do it will eventually get ugly then die
- https://www.youtube.com/watch?v=JoQN4xoXY5Y
- Quickly Analysing A Heap Memory Leak by Jack Shirazi



A methodology for approaching memory leaks

1. Do I have a leak (that needs fixing) ?
2. What is leaking (which classes) ?
3. What is keeping objects alive (an instance in the app) ?
4. Where is it leaking from (code where the objects are created and/or assigned) ?



Tools

- GC Logging
  - Suitable for production – GC logs remain after JVM terminates
- GCViewer
  - Suitable for production – views GC logs
- Heap Dumping
  - Suitable for production – but! freezes the JVM so only when necessary – log remains
- Eclipse MAT
  - Suitable for production – views Heap Dumps
- VisualVM (use 'profiler' with allocation stack traces recording on)
  - **NOT** Suitable for production – needs a live JVM and can crash it (all too often)

Class histogram

- `jmap -histo:live <pid>`
- Most profilers memory analysis histogram
- Heap dump histogram

Heap Dump

- `-XX:+HeapDumpOnOutOfMemoryError`
- `jmap -dump:live,file=<file-path> <pid>`
  - Or without "live," if you want to see dead objects that have not yet been GCed, "live," forces a GC before the dump
- JMX: com.sun.management.HotSpotDiagnostic.dumpHeap()
  - Eg from jconsole, visualvm, even programmatically
- `jcmd <pid> GC.heap_dump <file-path>`

Who am I? Jack Shirazi

- Working in Performance and Reliability Engineering Team at Hotels.com
  - Part of Expedia Group, handling $88billion in bookings 2017
- Founder of JavaPerformanceTuning.com
- Author of Java Performance Tuning (O'Reilly)
- Published over 60 articles on Java Performance Tuning & a monthly newsletter for 15 years & around 10 000 tuning tips
- Also researched Black Hole Thermodynamics & published papers on Protein Structure Prediction with the UKs largest Cancer Research organisation

#hcomtechnology – presenter: Jack Shirazi – – hotels.com – expedia.com – https://medium.com/hotels-com-technology 2

OS paging

- If your JVM pages during GC, the paging will dominate everything
- 10x-100x slower

SLO fails from CPU utilization

- Your CPU utilization is high and it's because the garbage collector threads are swamping the CPU when they kick in, starving your application threads of desired CPU
- Recent JVMs do understand container limits, but even so …
- With 1 vCPU, use Serial GC
- Otherwise, scale down the number of GC threads to vCPU-1 (or what makes sense for your application and the SLOs)
  - -Xgcthreads (OpenJ9)
  - -XX:ParallelGCThreads and -XX:ConcGCThreads (Hotspot)

SLO fails from startup time

- Startup time tends to be dominated by things other than GC. But if you get startup delayed by one or more very long GCs, just try a different collector
- Startup time is usually improved using class data sharing, partial AOTC restoring previous compilations, and tiered compilation (and making sure your app CAN start quickly).
  - These capabilities are best in the latest JVMs as startup time has become more of a priority in recent years
- Also try setting Xms to Xmx

SLO fails from footprint

- Footprint tuning is usually simple: lower Xmx until it's as low as possible while still achieving your SLOs (and use compressed oops if available)
- Some apps need a larger heap for spikes, but want the heap as small as possible when not dealing with spikes. For these, the best option is ParNew+SerialOld but that was discontinued in Java 8. For Java 11+
  - Serial will shrink best but the pauses may be unacceptable
  - Gencon has a good go at shrinking the heap
  - G1 with -XX:+UseStringDeduplication may give the best footprint

SLO fails from throughput

- Throughput priority is usually straightforward, and parallel or optthruput GCs will optimize your throughput
- However sometimes you prioritize throughput but still need a reasonable pause time. It's useful to know that actually the primary goal of parallel GC is pause time – just it's normally unset so that throughput gets optimized
  - -XX:MaxGCPauseMillis can be used, but you are directly decreasing throughput if you do, so you need to choose carefully
- Or try gencon which attempts to balance pauses and throughput

SLO fails from latency (pause time)

- Most GCs target minimizing pause time; these pause time targeting GCs use two strategies
    - Collect garbage incrementally while stopping the application
    - Collect garbage concurrently without stopping the application
- The smaller the pause time you want, the higher the overhead the GC will cost
- Pause time tends to be limited by
    - Compaction strategy
    - Live set scanning time

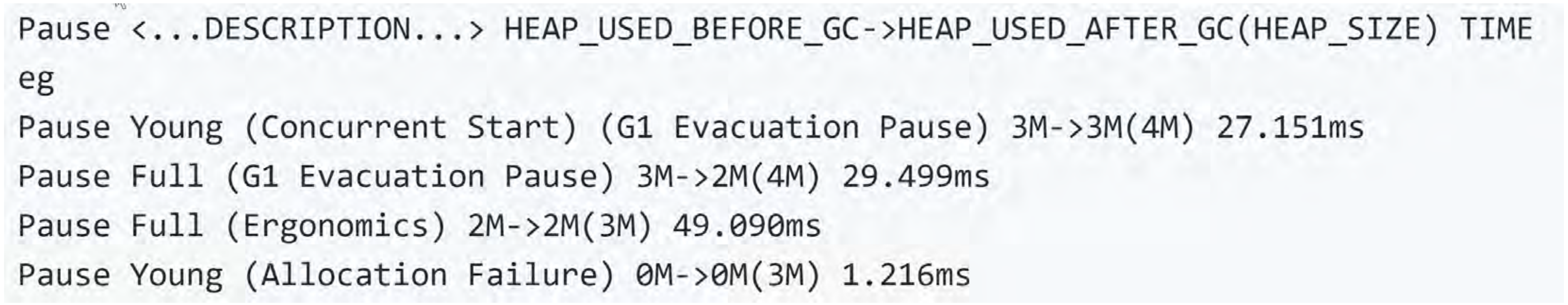
Other stuff

A couple of flags

- -XX:+DisableExplicitGC
  - If the code says System.gc() or equivalent somewhere, that flag will tell the GC to ignore the request
  - Explicitly calling the GC from code is a bad practice and typically causes suboptimal collection behaviour
- -XX:+PrintFlagsFinal
  - Very useful when diagnosing, to see what the configuration actually was rather than what someone thought it might have been

Monitoring pause times

- **Because pauses can be from other than the GC, it’s useful to log safepoint info too**
  - **-Xlog:safepoint*… (Java 8: -XX:+PrintGCApplicationStoppedTime)**
- **Pause times in GC logs are straightforward**



```
Pause <...DESCRIPTION...> HEAP_USED_BEFORE_GC->HEAP_USED_AFTER_GC(HEAP_SIZE) TIME
eg
Pause Young (Concurrent Start) (G1 Evacuation Pause) 3M->3M(4M) 27.151ms
Pause Full (G1 Evacuation Pause) 3M->2M(4M) 29.499ms
Pause Full (Ergonomics) 2M->2M(3M) 49.090ms
Pause Young (Allocation Failure) 0M->0M(3M) 1.216ms
```

Metaspace

- Metaspace stores the Java class metadata, the internal representation of Java class
- Allocated in Non-Heap native memory and can increase its size (up to what the OS provides)
- Limited by the JVM parameter "MaxMetaSpaceSize“
- Can cause a Full GC when it is full and needs expanding
- So size it avoid GCs

Code 1/3

- If you've tried the tuning flow and still don't achieve your SLOs, first lookup the best GC you found and try any obvious options (eg resizing the young gen, or starting GC collections earlier). But ultimately, the code may need tuning
- Start with looking for Finalizers and Reference processing in the GC logs, and if these take significant time
  - Eliminate Finalizers (this is a best practice anyway)
  - Reduce Reference object usage

Code 2/3

- Allocation rates limit how effective the GC can be. If you are allocating too fast, the GC can't keep up or, in the case of the recently built GCs, will slow down the allocations to let the GC keep up
- Either way, this impacts your app
- So reduce allocation rate by profiling allocations and targeting the top allocators
- A rule of thumb is up to 300MB/sec allocation should be okay for one of the GCs, 1GB/sec is too high

Code 3/3

- Big objects (typically large arrays) are problematic for garbage collectors, because they are expensive to move around in memory. So you want to avoid making them garbage
  - Try to pre-size collections to the maximum they will reach
  - Be aware of this cost for large collections that are temporary, they may be worth targeting if all else has failed
    - Some GCs will produce “humungous” object processing statistics to identify if these are a problem
  - Process streams directly, avoid intermediate copies of the stream data, or at worst using a small reusable buffer to process the data

Who Am I? Jack Shirazi

- In Reliability Engineering Team, Expedia Group
- Founder of JavaPerformanceTuning.com
- Author of Java Performance Tuning (O'Reilly)
- Published over 60 articles on Java Performance Tuning & a monthly newsletter for over 15 years & around 10 000 tuning tips