twitter

Attila Szegedi, Software Engineer
@asz

Everything I ever learned about JVM performance tuning @twitter

~~Everything~~ More than I ever wanted to learn~~ed~~ about JVM performance tuning @twitter

Twitter's biggest enemy

Twitter's biggest enemy

Latency

Latency contributors

- By far the biggest contributor is garbage collector
- others are, in no particular order:
  - in-process locking and thread scheduling,
  - I/O,
  - application algorithmic inefficiencies.

Areas of performance tuning

- Memory tuning
- Lock contention tuning
- CPU usage tuning
- I/O tuning

Areas of memory performance tuning

- Memory footprint tuning
- Allocation rate tuning
- Garbage collection tuning

Memory footprint tuning

- So you got an OutOfMemoryError…
- Maybe you just have too much data!
- Maybe your data representation is fat!
- You can also have a genuine memory leak…

Too much data

- Run with -verbosegc
- Observe numbers in "Full GC" messages
  `[Full GC $before->$after($total), $time secs]`

- Can you give the JVM more memory?
- Do you need all that data in memory? Consider using:
  - a LRU cache, or…
  - soft references*

Fat data

- Can be a problem when you want to do wacky things, like
  - load the full Twitter social graph in a single JVM
  - load all user metadata in a single JVM
- Slimming internal data representation works at these economies of scale

Fat data: object header

- JVM object header is normally two machine words.
- That's 16 bytes, or 128 bits on a 64-bit JVM!
- `new java.lang.Object()` takes 16 bytes.
- `new byte[0]` takes 24 bytes.

Fat data: padding

```
class A {
    byte x;
}
class B extends A {
    byte y;
}
```

- new A() takes 24 bytes.
- new B() takes 32 bytes.

Fat data: no inline structs

```
class C {
  Object obj = new Object();
}
```

- `new C()` takes 40 bytes.
- similarly, no inline array elements.

Slimming taken to extreme

- A research project had to load the full follower graph in memory
- Each vertex's edges ended up being represented as int arrays
- If it grows further, we can consider variable-length differential encoding in a byte array

Compressed object pointers

- Pointers become 4 bytes long
- Usable below 32 GB of max heap size
- Automatically used below 30 GB of max heap

Compressed object pointers

	Uncompressed	Compressed	32-bit
Pointer	8	4	4
Object header	16	12*	8
Array header	24	16	12
Superclass pad	8	4	4

* Object can have 4 bytes of fields and still only take up 16 bytes

Avoid instances of primitive wrappers

- Hard won experience with Scala 2.7.7:
  - a `Seq[Int]` stores `java.lang.Integer`
  - an `Array[Int]` stores `int`
  - first needs (24 + 32 * length) bytes
  - second needs (24 + 4 * length) bytes

Avoid instances of primitive wrappers

- This was fixed in Scala 2.8, but it shows that:
- you often don't know the performance characteristics of your libraries,
- and won't ever know them until you run your application under a profiler.

Map footprints

- Guava `MapMaker.makeMap()` takes 2272 bytes!
- `MapMaker.concurrencyLevel(1).makeMap()` takes 352 bytes!
- ConcurrentMap with level 1 makes sense sometimes (i.e. you don't want a ConcurrentModificationException)

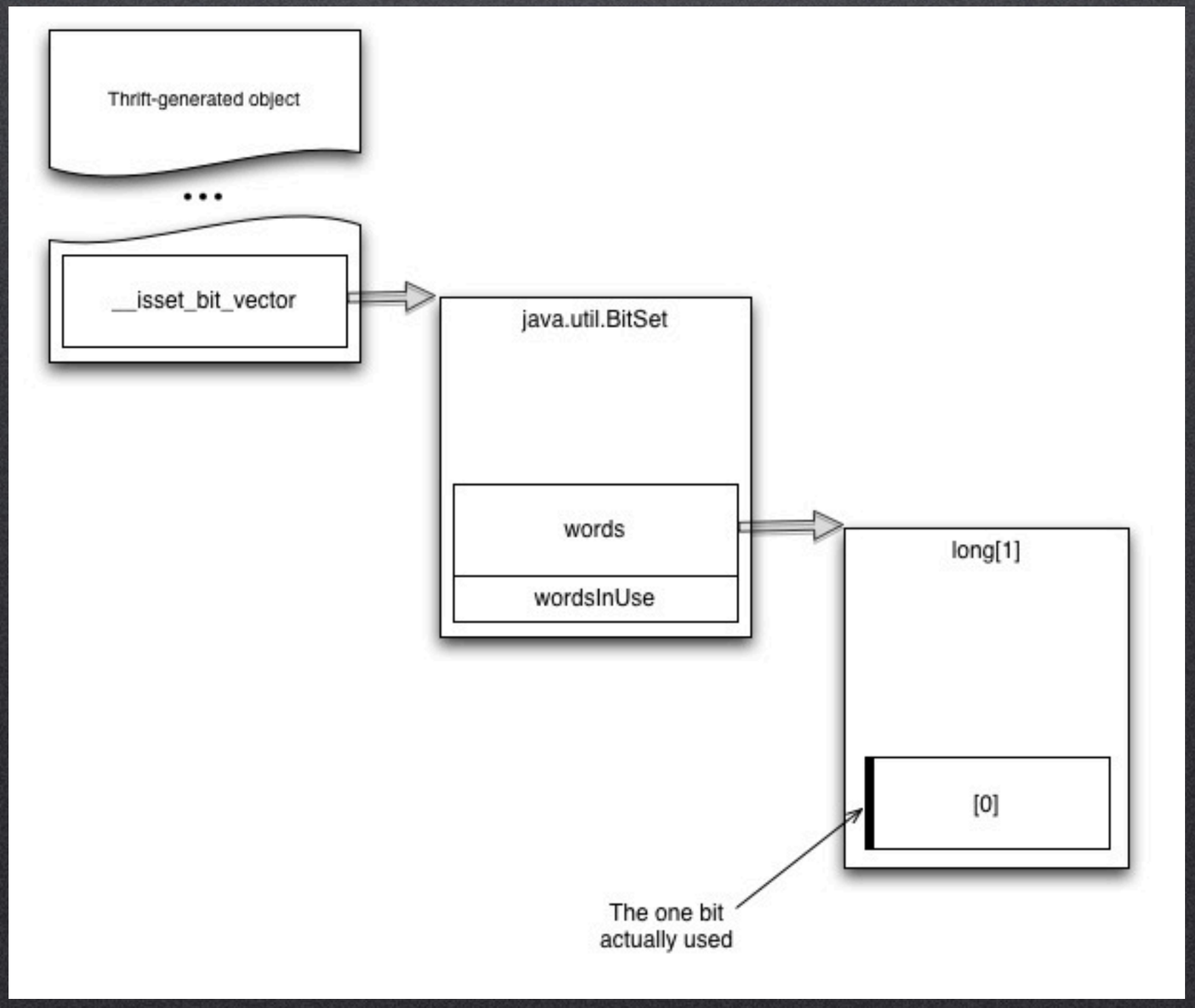
Thrift can be heavy

- Thrift generated classes are used to encapsulate a wire tranfer format.
- Using them as your domain objects: almost never a good idea.

Thrift can be heavy

- Every Thrift class with a primitive field has a `java.util.BitSet __isset_bit_vector` field.
- It adds between 52 and 72 bytes of overhead per object.

Thrift can be heavy



Thrift can be heavy

- Thrift does not support 32-bit floats.
- Coupling domain model with transport:
  - resistance to change domain model
- You also miss oportunities for interning and N-to-1 normalization.

```
class Location {
    public String city;
    public String region;
    public String countryCode;
    public int metro;
    public List<String> placeIds;
    public double lat;
    public double lon;
    public double confidence;
```

```
class SharedLocation {
    public String city;
    public String region;
    public String countryCode;
    public int metro;
    public List<String> placeIds;

class UniqueLocation {
    private SharedLocation sharedLocation;
    public double lat;
    public double lon;
    public double confidence;
```

Careful with thread locals

- Thread locals stick around.
- Particularly problematic in thread pools with m×n resource association.
  - 200 pooled threads using 50 connections: you end up with 10 000 connection buffers.
- Consider using synchronized objects, or
- just create new objects all the time.

Part II:
fighting latency

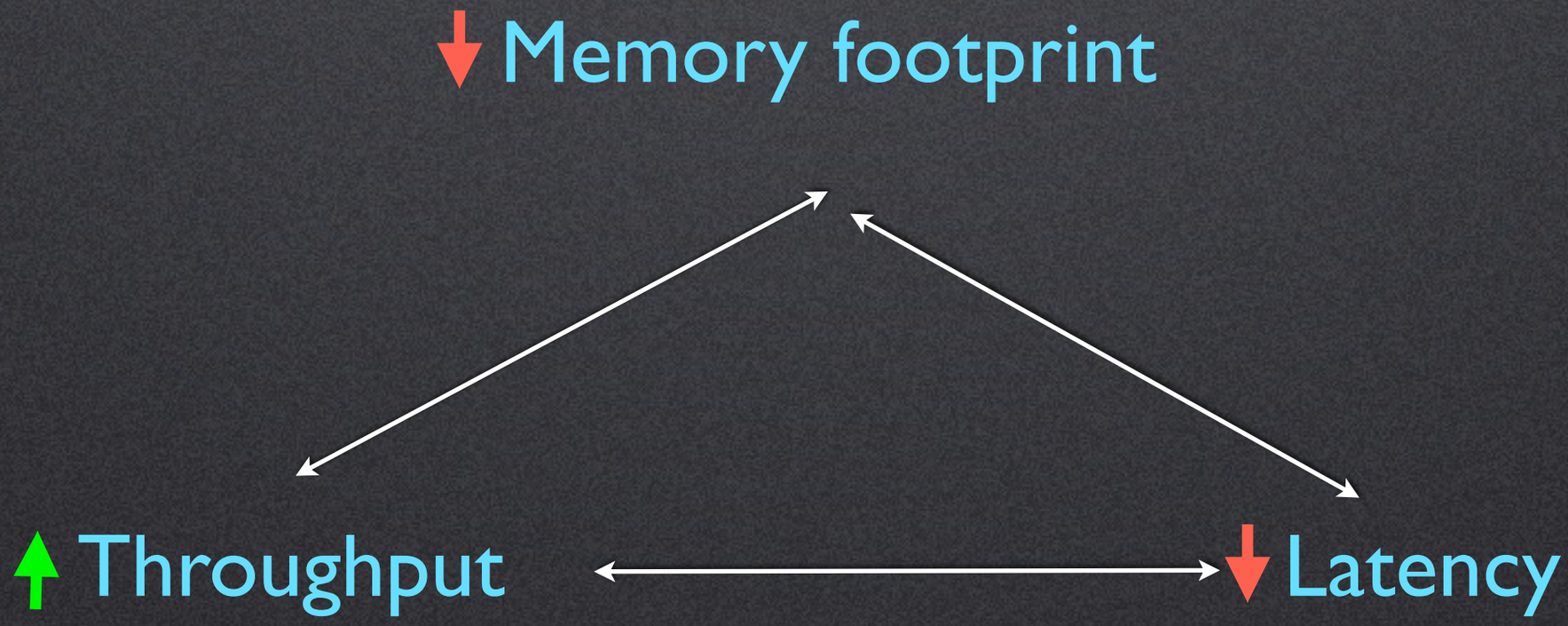
Performance tradeoff

Memory

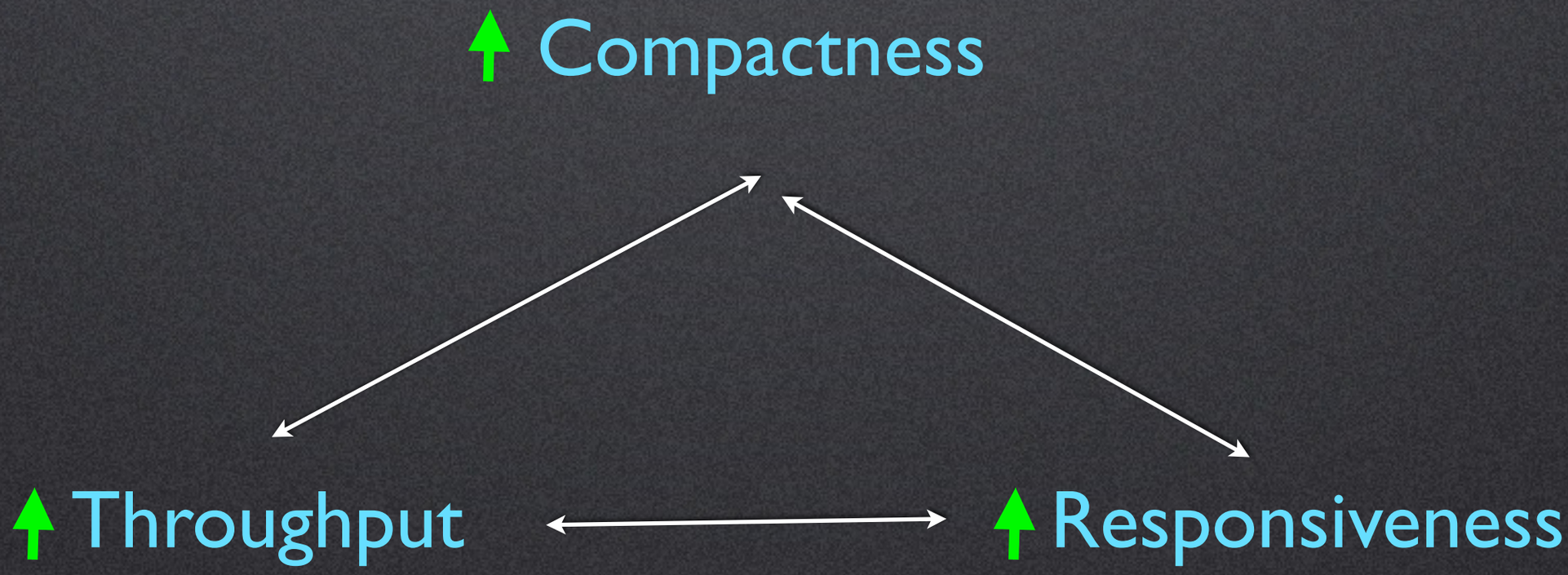
Time

Convenient, but oversimplified view.

Performance triangle



Performance triangle



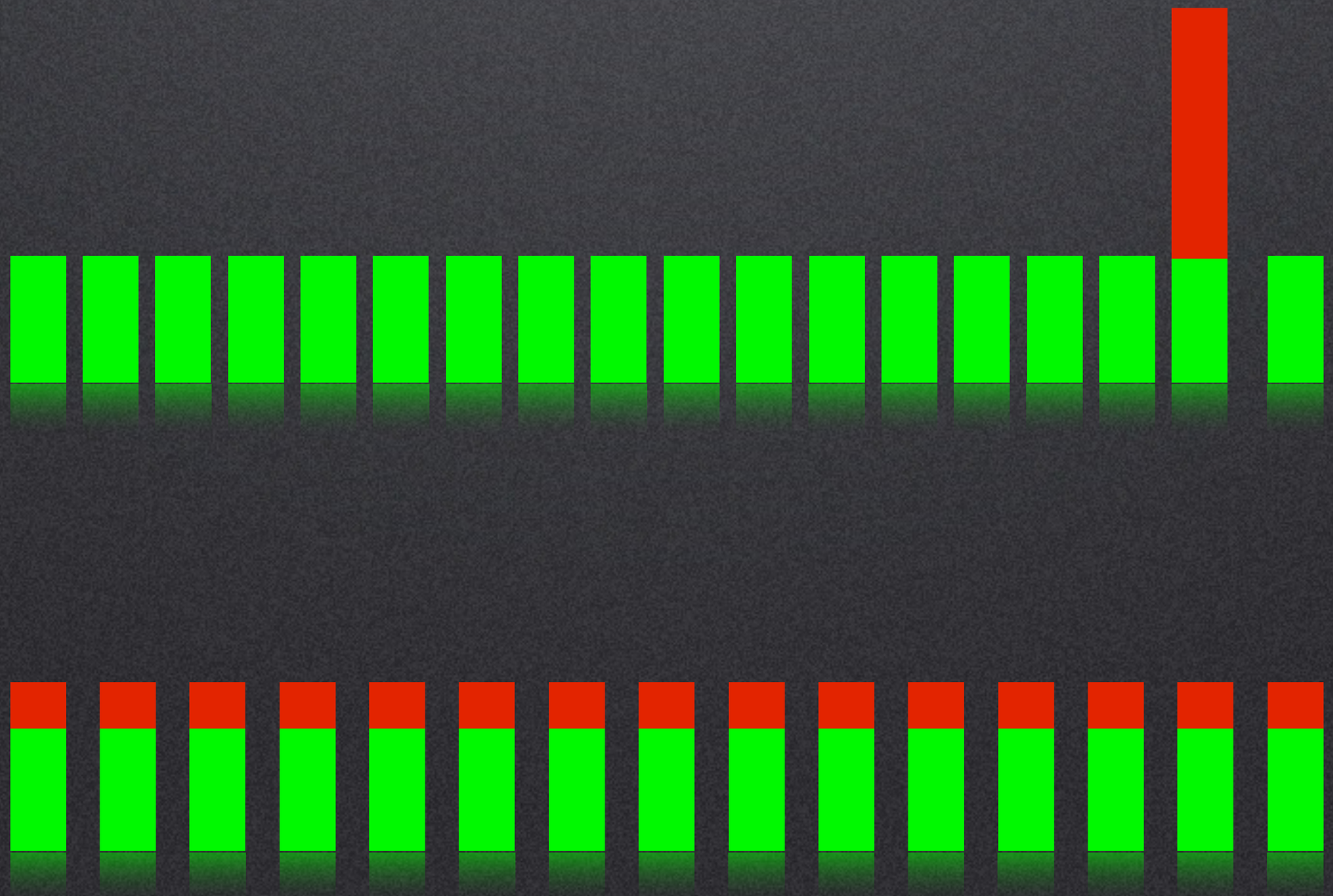
$$C \times T \times R = a$$

- Tuning: vary C, T, R for fixed a
- Optimization: increase a

Performance triangle

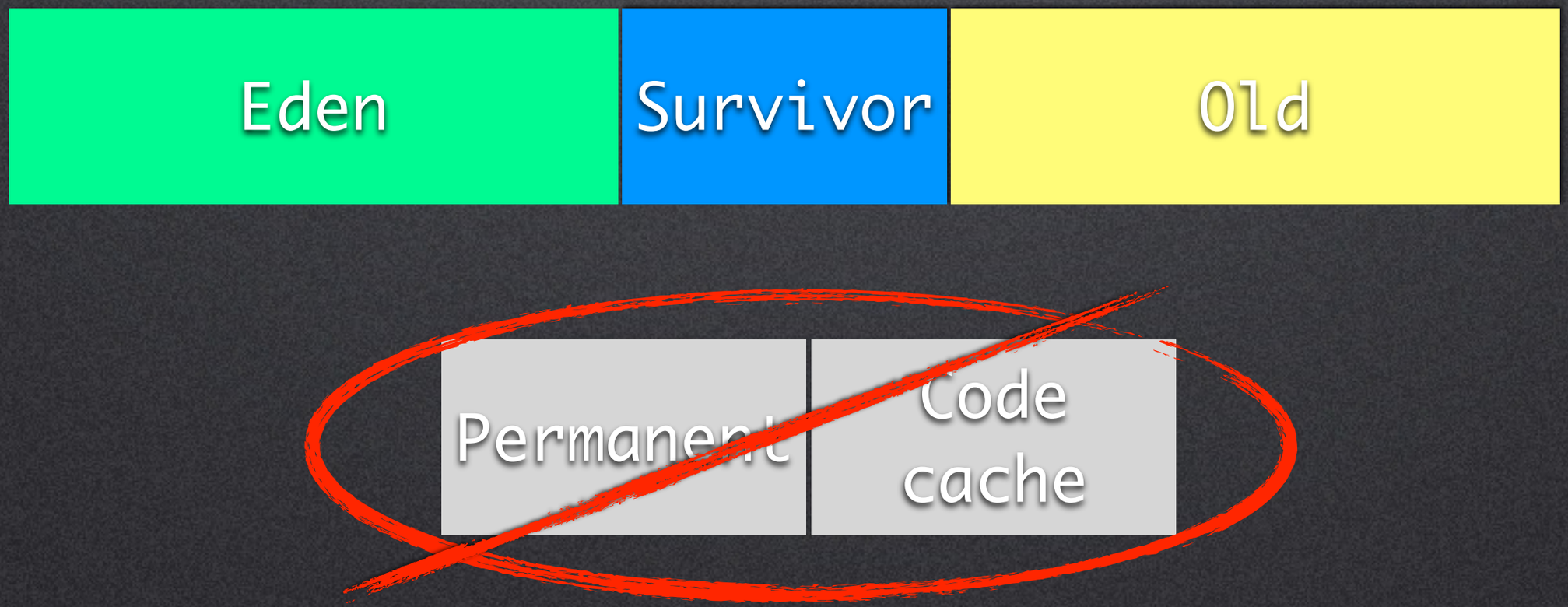
- Compactness: inverse of memory footprint
- Responsiveness: longest pause the application will experience
- Throughput: amount of useful application CPU work over time
- Can trade one for the other, within limits.
- If you have spare CPU, can be pure win.

Responsiveness vs. throughput



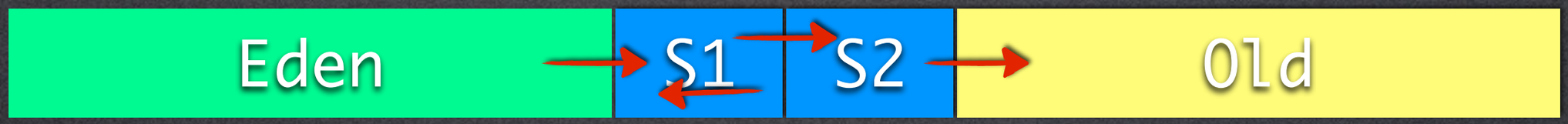
Biggest threat to responsiveness in the JVM is the garbage collector

Memory pools



This is entirely HotSpot specific!

How does young gen work?



- All new allocation happens in eden.
  - It only costs a pointer bump.
- When eden fills up, stop-the-world copy-collection into the survivor space.
  - Dead objects cost zero to collect.
- Aftr several collections, survivors get tenured into old generation.

Ideal young gen operation

- Big enough to hold more than one set of all concurrent request-response cycle objects.
- Each survivor space big enough to hold active request objects + tenuring ones.
- Tenuring threshold such that long-lived objects tenure fast.

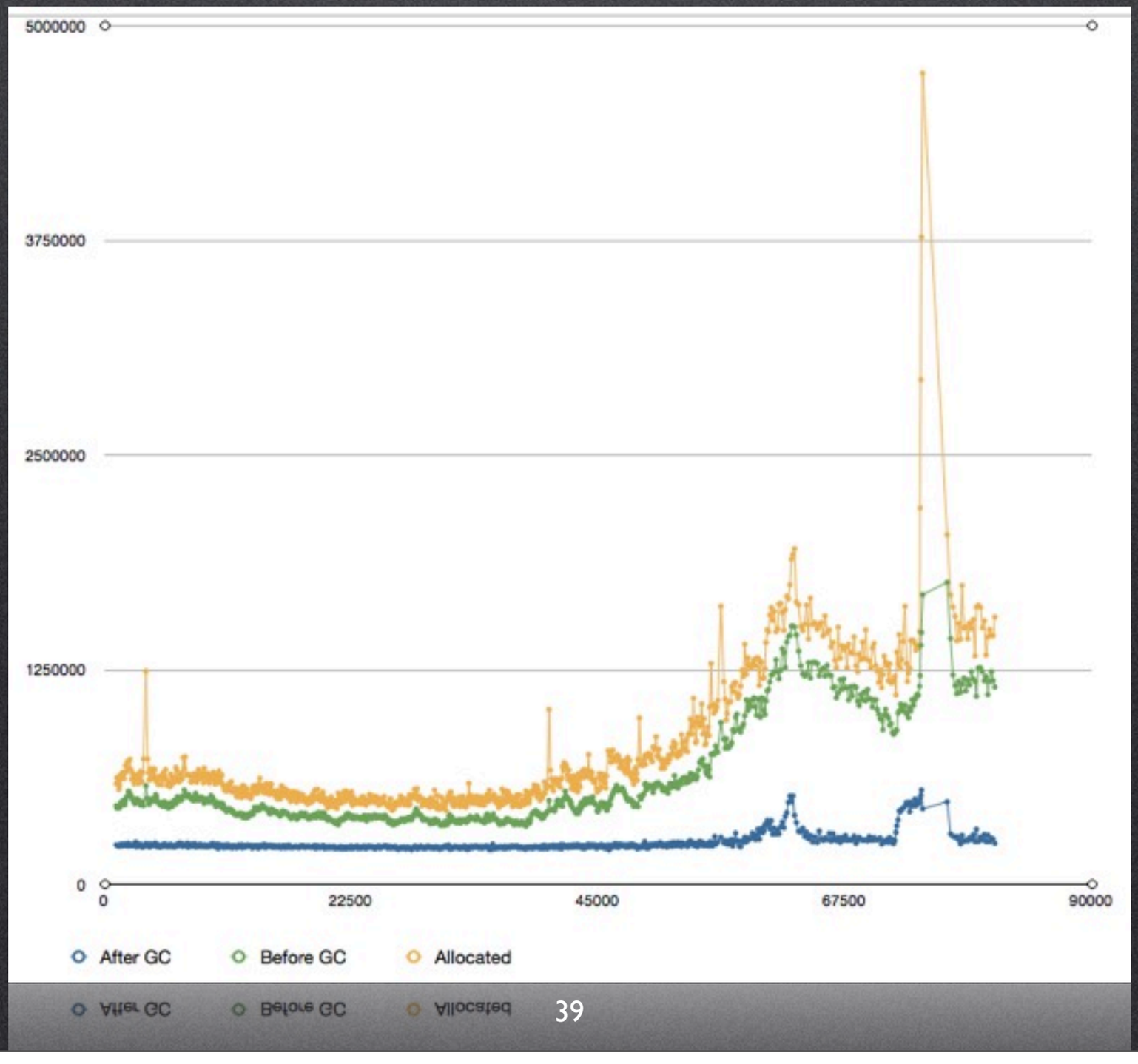
Old generation collectors

- Throughput collectors
  - `-XX:+UseSerialGC`
  - `-XX:+UseParallelGC`
  - `-XX:+UseParallelOldGC`
- Low-pause collectors
  - `-XX:+UseConcMarkSweepGC`
  - `-XX:+UseG1GC` (can't discuss it here)

Adaptive sizing policy

- Throughput collectors can automatically tune themselves:
  - -XX:+UseAdaptiveSizePolicy
  - -XX:MaxGCPauseMillis=… (i.e. 100)
  - -XX:GCTimeRatio=… (i.e. 19)

Adaptive sizing policy at work



Choose a collector

- Bulk service: throughput collector, no adaptive sizing policy.
- Everything else: try throughput collector with adaptive sizing policy. If it didn't work, use concurrent mark-and-sweep (CMS).

Always start with tuning the young generation

- Enable `-XX:+PrintGCDetails`, `-XX:+PrintHeapAtGC`, and `-XX:+PrintTenuringDistribution`.
- Watch survivor sizes! You'll need to determine "desired survivor size".
- There's no such thing as a "desired eden size", mind you. The bigger, the better, with some responsiveness caveats.
- Watch the tenuring threshold; might need to tune it to tenure long lived objects faster.

-XX:+PrintHeapAtGC

```
Heap after GC invocations=7000 (full 87):
 par new generation   total 4608000K, used 398455K
  eden space 4096000K,   0% used
  from space 512000K,  77% used
  to   space 512000K,   0% used
 concurrent mark-sweep generation total 3072000K, used 1565157K
 concurrent-mark-sweep perm gen total 53256K, used 31889K
}
```

-XX:+PrintTenuringDistribution

```
Desired survivor size 262144000 bytes, new threshold 4 (max 4)
- age   1:  137474336 bytes,  137474336 total
- age   2:   37725496 bytes,  175199832 total
- age   3:   23551752 bytes,  198751584 total
- age   4:   14772272 bytes,  213523856 total
```

- Things of interest:
  - Number of ages
  - Size distribution in ages
    - You want strongly declining.

Tuning the CMS

- Give your app as much memory as possible.
  - CMS is speculative. More memory, less punitive miscalculations.
- Try using CMS without tuning. Use `-verbosegc` and `-XX:+PrintGCDetails`.
  - Didn't get any "Full GC" messages? You're done!
- Otherwise, tune the young generation first.

Tuning the old generation

- Goals:
  - Keep the fragmentation low.
  - Avoid full GC stops.
- Fortunately, the two goals are not conflicting.

Tuning the old generation

- Find the minimum and maximum working set size (observe "Full GC" numbers under stable state and under load).
- Overprovision the numbers by 25-33%.
  - This gives CMS a cushion to concurrently clean memory as it's used.

Tuning the old generation

- Set `-XX:InitiatingOccupancyFraction` to between 80-75, respectively.
  - corresponds to overprovisioned heap ratio.
- You can lower initiating occupancy fraction to 0 if you have CPU to spare.

Responsiveness still not good enough?

- Too many live objects during young gen GC:
  - Reduce NewSize, reduce survivor spaces, reduce tenuring threshold.
- Too many threads:
  - Find the minimal concurrency level, or
  - split the service into several JVMs.

Responsiveness still not good enough?

- Does the CMS abortable preclean phase, well, abort?
- It is sensitive to number of objects in the new generation, so
  - go for smaller new generation
  - try to reduce the amount of short-lived garbage your app creates.

Part III:
let's take a break from GC

Thread coordination optimization

- You don't have to always go for `synchronized`.
- Synchronization is a read barrier on entry; write barrier on exit.
- Sometimes you only need a half-barrier; i.e. in a producer-observer pattern.
- Volatiles can be used as half-barriers.

Thread coordination optimization

- For atomic update of a single value, you only need `Atomic{Integer|Long}.compareAndSet()`.
- You can use `AtomicReference.compareAndSet()` for atomic update of composite values represented by immutable objects.

Fight CMS fragmentation with slab allocators

- CMS doesn't compact, so it's prone to fragmentation, which will lead to a stop-the-world pause.
- Apache Cassandra uses a slab allocator internally.

Cassandra slab allocator

- 2MB slab sizes
- copy byte[] into them using compare-and-set
- GC before: 30-60 seconds every hour
- GC after: 5 seconds once in 3 days and 10 hours

Slab allocator constraints

- Works for limited usage:
  - Buffers are written to linearly, flushed to disk and recycled when they fill up.
    - The objects need to be converted to binary representation anyway.
- If you need random freeing and compaction, you're heading down the wrong direction.
- If you find yourself writing a full memory manager on top of byte buffers, stop!

Soft references revisited

- Soft reference clearing is based on the amount of free memory available when GC encounters the reference.
- By definition, throughput collectors always clear them.
- Can use them with CMS, but they increase memory pressure and make the behavior less predictable.
- Need two GC cycles to get rid of referenced objects.

~~Everything~~ More than I ever wanted to learn~~ed~~ about JVM performance tuning @twitter

Questions?

twitter.com/jobs
@jointheflock

twitter 

Attila Szegedi, Software Engineer
@asz