Frugal memory management on the JVM

Galo Navarro
@srvaroa

#SchTechTalks

Guillermo Ontañón
@gontanon

Agenda

- Assuming basic Java / JVM knowledge
- Understanding our app’s workload / footprint
- Tradeoffs to address some of the common pathologies

Memory management on the JVM

Pros:

- Fast allocation / release of memory
- Can optimize on runtime

But if left unchecked may also cause:

- Big latency spikes / jitter
- High memory footprint

jstat -gc

E: Eden
O: Old gen
M: Metaspace
S: Survivor

C: capacity
U: utilisation

Y: Young GC
F: Full GG
CCS: Compressed class space

T: Time

```
jstat -gc 3441 500
```

S0C	S1C	S0U	S1U	EC	EU	OC	OU	MC	MU	CCSC	CCSU	YGC	YGCT	FGC	FGCT	GCT
512,0	512,0	32,0	0,0	1705984,0	204863,2	412160,0	294957,4	21552,0	12570,9	4144,0	1122,7	137682	332,212	2	0,159	332,371
512,0	512,0	32,0	0,0	1960448,0	117701,7	412160,0	294957,4	21552,0	12570,9	4144,0	1122,7	137684	332,217	2	0,159	332,376
512,0	512,0	32,0	0,0	2253312,0	0,0	412160,0	294957,4	21552,	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	82
512,0	512,0	0,0	32,0	2157568,0	1294911,1	412160,0	294957,4	21552	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	384
512,0	512,0	0,0	32,0	1979904,0	752715,0	412160,0	294957,4	21552,	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	90
512,0	512,0	0,0	32,0	1819136,0	327692,9	412160,0	294957,4	21552,	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	94
512,0	512,0	0,0	32,0	1673216,0	0,0	412160,0	294957,4	21552,	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	99
512,0	512,0	0,0	32,0	1541120,0	215881,7	412160,0	294957,4	21552,	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	04
512,0	512,0	0,0	32,0	2216960,0	443453,5	412160,0	294957,4	21552,	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	09
512,0	512,0	0,0	32,0	2033152,0	0,0	412160,0	294957,4	21552,	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	14
512,0	512,0	32,0	0,0	1948160,0	1442458,9	412160,0	294957,4	21552	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	417
512,0	512,0	32,0	0,0	1789952,0	1253723,3	412160,0	294957,4	21552	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	421
512,0	512,0	32,0	0,0	1647104,0	1351500,6	412160,0	294957,4	21552	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	426
512,0	512,0	0,0	32,0	1817600,0	0,0	412160,0	294957,4	21552,0	12570,9	4144,0	1122,7	137707	332,274	2	0,159	332,434
512,0	512,0	0,0	32,0	1671680,0	0,0	412160,0	294957,4	21552,0	12570,9	4144,0	1122,7	137709	332,278	2	0,159	332,438
512,0	512,0	0,0	32,0	1539584,0	339015,7	412160,0	294957,4	21552,0	12570,9	4144,0	1122,7	137711	332,283	2	0,159	332,442

Assume generational hypothesis, and expect that most objects will be orphaned soon

GC logs

```
-Xloggc:$PATH                               consider ramdisk / ssd [1]
-XX:+PrintGCDetails                     detailed GC logging
-XX:+PrintTenuringDistribution          aages
-XX:+PrintPromotionFailure
-XX:+PrintGCApplicationStoppedTimepauses, GC but also safepoints...
-XX:+PrintAdaptiveSizePolicy                ergonomic decisions

-XX:+UseGCLogFileRotation
-XX:NumberOfGCLogFiles=$NUM_FILES           default 1
-XX:GCLogFileSize=$SIZE[M|K]                default 512k
```

```
2016-02-25T11:58:21.628+0800: [GC pause (G1 Evacuation Pause) (young)
Desired survivor size 8053063680 bytes, new threshold 4 (max 15)
- age   1:  609830392 bytes,  609830392 total
- age   2:  635249376 bytes, 1245079768 total
- age   3:  530928792 bytes, 1776008560 total
- age   4: 6566883776 bytes, 8342892336 total
- age   5:  160917504 bytes, 8503809840 total
, 2.3754150 secs]
   [Parallel Time: 2305.9 ms, GC Workers: 23]
      [GC Worker Start (ms): Min: 63546061.8, Avg: 63546062.1, Max: 63546062.4, Diff: 0.6]
      [Ext Root Scanning (ms): Min: 0.0, Avg: 0.3, Max: 0.6, Diff: 0.5, Sum: 6.3]
      [SATB Filtering (ms): Min: 0.0, Avg: 0.0, Max: 0.1, Diff: 0.1, Sum: 0.1]
      [Update RS (ms): Min: 16.7, Avg: 18.3, Max: 22.8, Diff: 6.1, Sum: 420.7]
         [Processed Buffers: Min: 5, Avg: 10.4, Max: 18, Diff: 13, Sum: 239]
      [Scan RS (ms): Min: 247.8, Avg: 252.3, Max: 253.9, Diff: 6.1, Sum: 5803.5]
      [Code Root Scanning (ms): Min: 0.0, Avg: 0.0, Max: 0.0, Diff: 0.0, Sum: 0.3]
      [Object Copy (ms): Min: 2032.6, Avg: 2033.1, Max: 2034.4, Diff: 1.8, Sum: 46762.1]
      [Termination (ms): Min: 0.0, Avg: 1.3, Max: 1.7, Diff: 1.7, Sum: 30.3]
      [GC Worker Other (ms): Min: 0.0, Avg: 0.1, Max: 0.2, Diff: 0.2, Sum: 2.1]
      [GC Worker Total (ms): Min: 2305.2, Avg: 2305.5, Max: 2305.8, Diff: 0.6, Sum: 53025.4]
      [GC Worker End (ms): Min: 63548367.5, Avg: 63548367.6, Max: 63548367.7, Diff: 0.2]
   [Code Root Fixup: 0.1 ms]
   [Code Root Migration: 0.1 ms]
   [Code Root Purge: 0.0 ms]
   [Clear CT: 2.1 ms]
   [Other: 67.2 ms]
      [Choose CSet: 0.0 ms]
      [Ref Proc: 0.5 ms]
      [Ref Enq: 0.0 ms]
      [Redirty Cards: 62.2 ms]
      [Free CSet: 2.8 ms]
   [Eden: 21.9G(21.9G)->0.0B(27.2G) Survivors: 8288.0M->2848.0M Heap: 76.0G(94.0G)->57.3G(95.2G)]
 [Times: user=53.07 sys=0.05, real=2.37 secs]
```

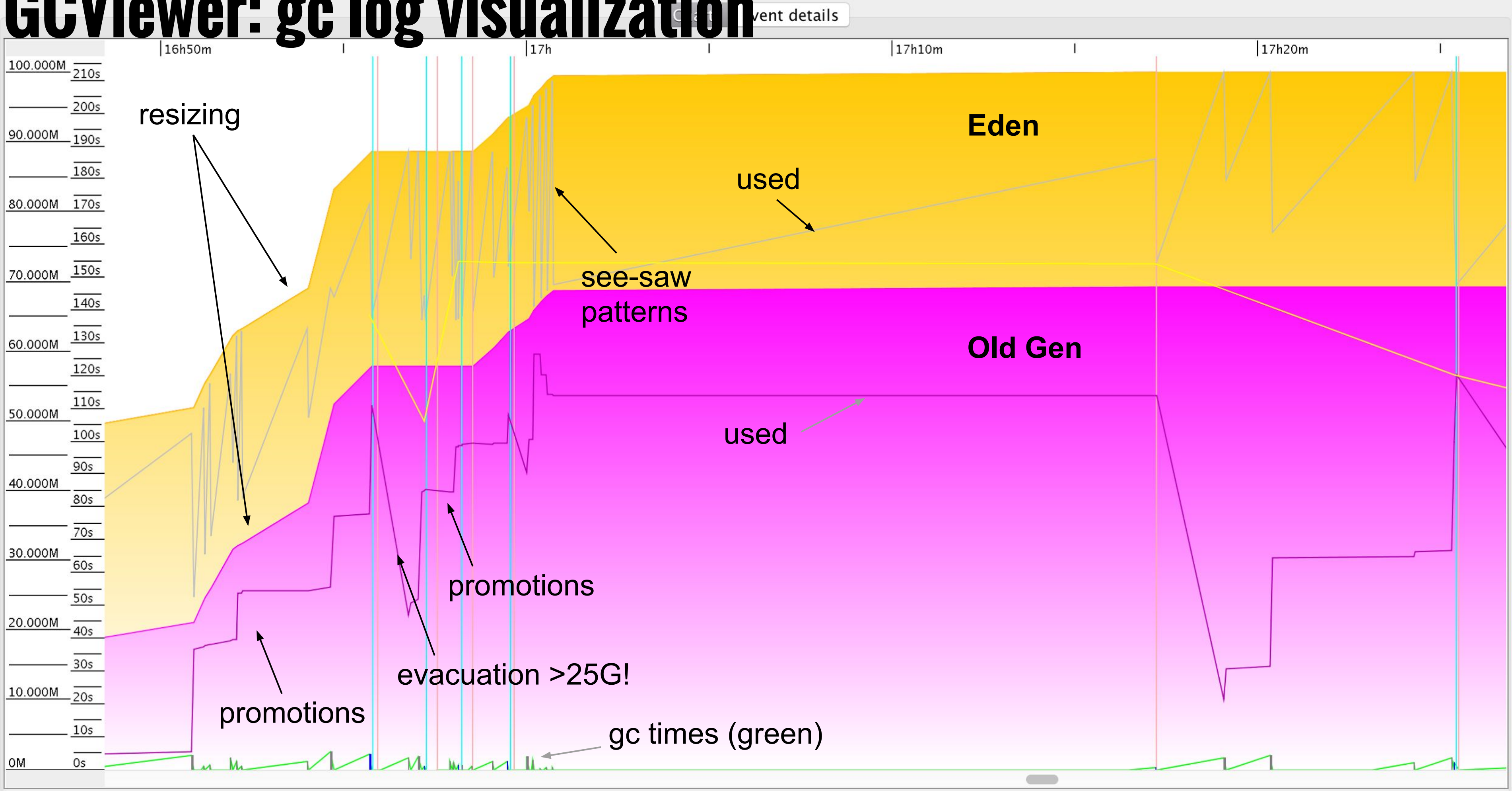
Assume generational hypothesis, and expect most objects to be dereferenced soon

Easy to correlate to service latency metrics

```
2016-02-25T11:14:59.233+0800: [GC pause (G1 Humongous Allocation) (young) (initial-mark)
Desired survivor size 8053063680 bytes, new threshold 1 (max 15)
- age   1: 9474955328 bytes, 9474955328 total
- age   2: 6322525168 bytes, 15797480496 total
- age   3:  176071416 bytes, 15973551912 total
- age   4:  132526584 bytes, 16106078496 total
, 5.1656688 secs]
   [Parallel Time: 5102.7 ms, GC Workers: 23]
      [GC Worker Start (ms): Min: 60943668.2, Avg: 60943668.6, Max: 60943668.9, Diff: 0.7]
      [Ext Root Scanning (ms): Min: 0.0, Avg: 0.3, Max: 2.5, Diff: 2.5, Sum: 6.2]
      [Code Root Marking (ms): Min: 0.0, Avg: 0.3, Max: 3.1, Diff: 3.1, Sum: 6.3]
      [Update RS (ms): Min: 13.9, Avg: 17.0, Max: 19.0, Diff: 5.1, Sum: 392.1]
         [Processed Buffers: Min: 7, Avg: 10.3, Max: 16, Diff: 9, Sum: 238]
      [Scan RS (ms): Min: 301.0, Avg: 302.7, Max: 303.6, Diff: 2.6, Sum: 6962.2]
      [Code Root Scanning (ms): Min: 0.0, Avg: 0.0, Max: 0.0, Diff: 0.0, Sum: 0.3]
      [Object Copy (ms): Min: 4781.1, Avg: 4781.7, Max: 4782.4, Diff: 1.3, Sum: 109978.1]
      [Termination (ms): Min: 0.0, Avg: 0.1, Max: 0.1, Diff: 0.1, Sum: 1.4]
      [GC Worker Other (ms): Min: 0.0, Avg: 0.1, Max: 0.2, Diff: 0.1, Sum: 1.9]
      [GC Worker Total (ms): Min: 5101.8, Avg: 5102.1, Max: 5102.5, Diff: 0.7, Sum: 117348.6]
      [GC Worker End (ms): Min: 60948770.7, Avg: 60948770.7, Max: 60948770.8, Diff: 0.1]
   [Code Root Fixup: 0.0 ms]
   [Code Root Migration: 0.1 ms]
   [Code Root Purge: 0.0 ms]
   [Clear CT: 2.7 ms]
   [Other: 60.0 ms]
      [Choose CSet: 0.0 ms]
      [Ref Proc: 1.2 ms]
      [Ref Enq: 0.0 ms]
      [Redirty Cards: 52.2 ms]
      [Free CSet: 4.6 ms]
   [Eden: 13.0G(15.0G)->0.0B(23.1G) Survivors: 15.0G->7040.0M Heap: 63.9G(81.4G)->58.0G(86.7G)]
 [Times: user=115.69 sys=1.55, real=5.16 secs]
```

Different cause

GCViewer: gc log visualization



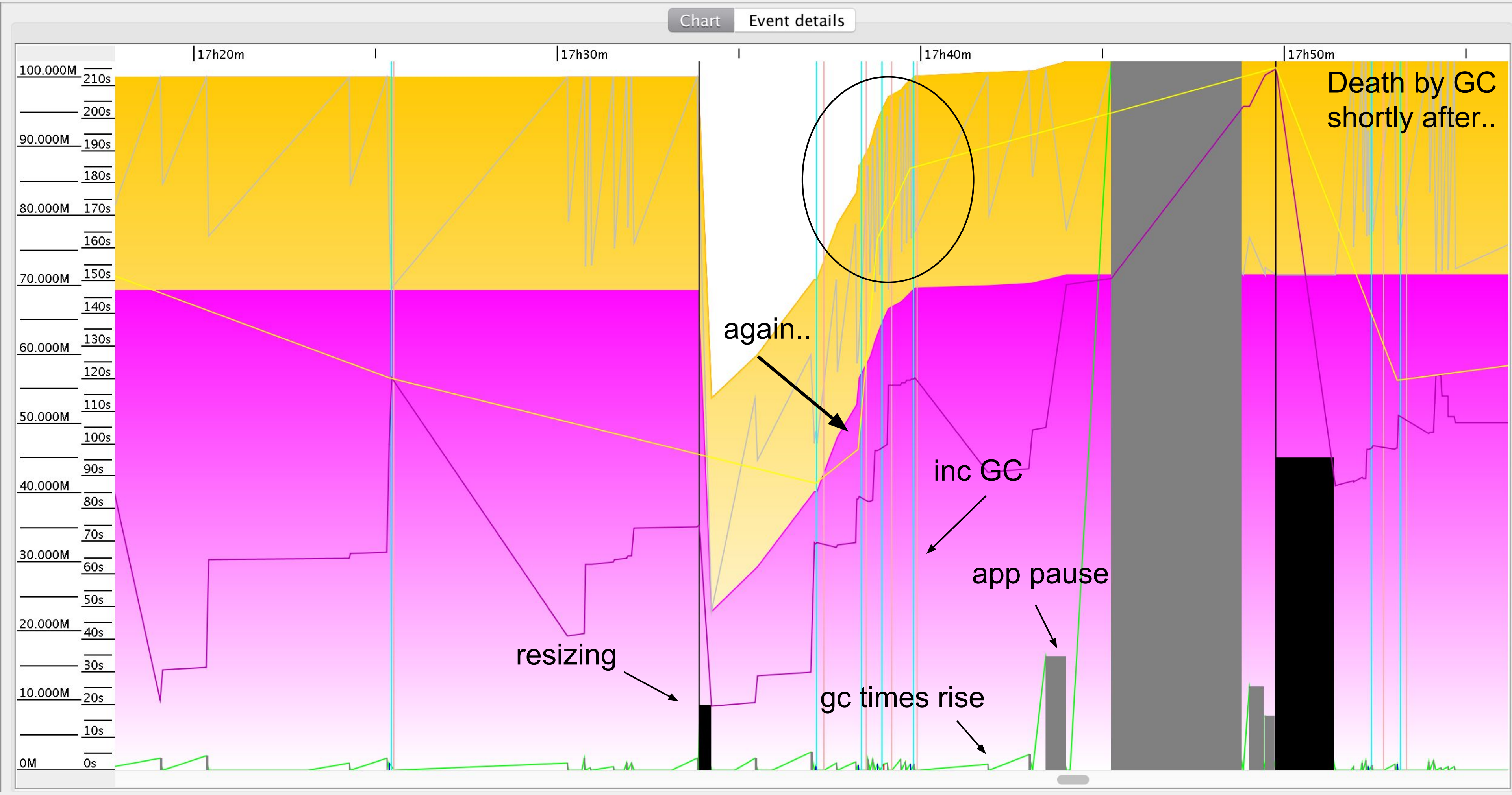


Chart
Event details
17h20m
17h30m
17h40m
17h50m
100.000M
90.000M
80.000M
70.000M
60.000M
50.000M
40.000M
30.000M
20.000M
10.000M
0M
210s
200s
190s
180s
170s
160s
150s
140s
130s
120s
110s
100s
90s
80s
70s
60s
50s
40s
30s
20s
10s
0s
Death by GC shortly after..
again..
inc GC
app pause
resizing
gc times rise

What is in the heap?

```
num     #instances         #bytes  class name
----------------------------------------------
  1:     456735295    29968183048  [C
  2:     141993549    17650832184  [Ljava.lang.Object;
  3:     432874195    13851974240  java.lang.String
  4:     141783960     5671358400  java.util.ArrayList
  5:     220867901     5300829624  java.lang.Long
  6:       3507261     3992725000  [I
  7:      90242360     3839836936  [B
  8:     142089373     3410144952                    .JDBCRecord
```

```
num     #instances         #bytes  class name
----------------------------------------------
  1:     309717286    18357971096  [C
  2:     103220845    12801064160  [Ljava.lang.Object;
  3:     309717154     9910948928  java.lang.String
  4:     103219696     4128787840  java.util.ArrayList
  5:     103216209     2477189016                    .JDBCRecord
  6:     103005153     2472123672  java.lang.Long
  7:          5741       22010304  [B
  8:        211348        5072352  java.lang.Double
```

```
jmap -histo $PID
jmap -histo:live $PID -> triggers GC
```

```
class JDBCRecord {

    private List<Object> = ..
    private Long timestamp = ..
    private String ..
    private String ..
    private String ..

}
```

Profiling allocation rate: JMH

http://openjdk.java.net/projects/code-tools/jmh/

```
Options opt = new OptionsBuilder()
    .include(MyBenchmark.class.getSimpleName())
    .warmupIterations(5)
    .verbosity(VerboseMode.EXTRA)
    .addProfiler(HotspotRuntimeProfiler.class)
    .addProfiler(GCProfiler.class)
    .build();
new Runner(opt).run();
```

Profiling allocation rate: JMH

```
Iteration   9: 1464,198 ops/ms
    consumer16:                     779,567 ops/ms
    producer16:                     684,632 ops/ms
    ·gc.alloc.rate:                 60,088 MB/sec
    ·gc.alloc.rate.norm:            42,995 B/op
    ·gc.churn.PS_Eden_Space:        94,241 MB/sec
    ·gc.churn.PS_Eden_Space.norm:     67,432 B/op
    ·gc.count:                      1,000 counts
    ·gc.time:                       661,000 ms
```

Useful specially for small-ish sections of the fast path

Best practices

Costs of abstraction

Footprint aware data modelling

Object headers

http://hg.openjdk.java.net/jdk8/jdk8/hotspot/file/tip/src/share/vm/oops/oop.hpp
http://hg.openjdk.java.net/jdk8/jdk8/hotspot/file/tip/src/share/vm/oops/markOop.hpp

- 64-bit: 12 bytes padded to multiple of 8 → 16 bytes
- 32-bit: 8 bytes padded to multiple of 4 → 12 bytes

References

- Ref = 4 bytes on < 32G heaps
- Ref = 8 bytes on 64-bit JVMs with >32G heaps

Arrays: 1 ref to type, 4 bytes for length, 1 ref per element. Min 8/16 bytes

Boxing

`long`: 8 bytes → `Long`: 8 + 16 → 24 bytes (x3)

`boolean`: 1 bit → `Boolean`: 1 byte + 12 + 3 bytes padding → 16 bytes (x128)

Real world example: music license cache

20M song catalogue, ~200 countries, 9 types of perms

At 16 bytes per flag = 536 GiB → dataset split in N servers

At 1 bytes per flag = 34 GiB → dataset in 1 server

Boxing

Avoid using primitive types, pack flags in bytes or `BitSet`:

```
class Event {
    private long timestamp = ..
    private byte flags = ..
}
```

Scala: value types - very limited: only 1 val & defs, no inheritance, no initialization, sometimes allocates

```
class Counter(val underlying: Long) extends AnyVal {
  def inc: Wrapper = new Counter(underlying + 1)
}
```

Java 10: Project Valhalla brings value types "*Codes like a class, works like an int!*"

Fat data model

```
while ((line = reader.readLine()) != null) {
      users.add(new User(line));
}

class User {
      private final String name;
      private final Date birth;
      …
      public User(String s) {
            String[] fields = s.split("::");
            this.name = fields[0];
            this.birth = dateFormat.parse(fields[1]);
            ...
      }
      public String getName() { .. }
      public Date getBirth() { return  new Date(birth.getTime) }
      ...
      public String getXXX()
}
```

Good OOP, trying to save CPU on access.. but..

Can we afford multiplying dataset sizes?

Does our internal representation need to mirror the public contract?

Lazy parsing

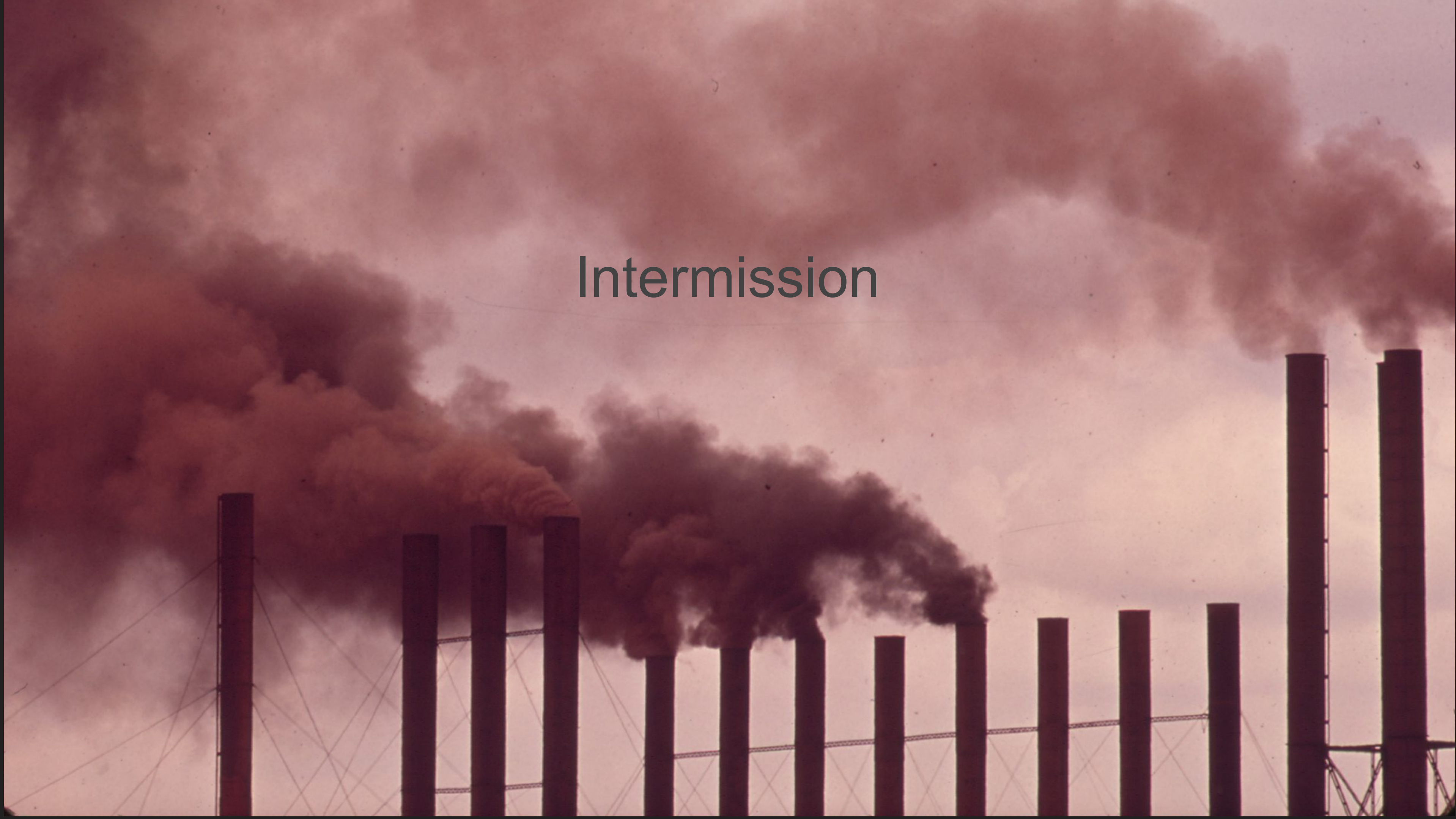
```
class User {

  private final String data;
  private volatile Date date = null;

  public Date getBirth() {
    if (date == null) {
      this.date = new Date(findField(1))
    }
    return date;
  }
  ...
  private final String findField(int n) {
    // loop to find field
    ...
  }
```

Might make sense.. (or, store offsets but not parse) to delay allocation until it's really needed

- Useful in hashMaps
- Think more complex cases (e.g.: network packets)


Intermission

When / where to remove allocations

You’ve already fixed the low hanging fruit (boxing, logs, ...)

GC has been tuned

Your application processes 1000s of QPS, latency sensitive

You have allocation and GC churn, latency and latency jitter are too high

Goal: less (zero?) allocations in your fast path

Techniques

Most are intrusive, lots of work to retrofit

Before optimizing something:

- JVM optimizes stuff: prove garbage with profiler / microbenchmark
- It’s in your fast path

It’s better to know in advance and design up front

Instant throwaway objects

They are objects that you create on the spot and immediately discard

Essentially: local variables

In C++, you'd declare them on the stack, the JVM does 'escape analysis'

What do do? Promote to instance members and reuse

IMPORTANT: method becomes thread unsafe unless you use a `ThreadLocal`

Escape analysis

```
public class A {
  private final int x;
  public A(final int _x) {
    this.x = _x;
  }
  public int getX() { return x; }
}

public void f(int n) {
   A a = new A(i);
   System.out.println(a.getX());
}
```

Likely JIT'd version

```
public void f(int n) {
  int _x = x;
  System.out.println(_x);
}
```

Objects that don't escape the current method or thread might get stack allocation

Disable with `-XX:-DoEscapeAnalysis`, to compare behaviour

Collections with per-request churn

Lists → always use array-backed lists, `ArrayList`

Linked lists, trees → Build your own intrusive implementations

Maps, Sets → Each insertion creates `Map.Entry`. Move off-process or off-heap

Queues → array backed, bounded (also gives you back pressure)

Primitive type collections → avoid boxing by using specialized implementations

Collections

Primitive types, zero allocations...

Trove → http://trove.starlight-systems.com/ (GPL)
OpenHFT → https://github.com/OpenHFT (Apache)

Off-heap implementations exist (note: have not tried them):

MapDB → www.mapdb.org/
OpenHFT

Off-process

Beware: the client library for your caching system may not be allocation free

Interning

The JVM does this for some objects

i.e. numeric vals. < 128, `String.intern()`

The set of possible values of an immutable object is known / manageable

Objects can be constructed from a key composed of primitive types

Keep them cached in a hash table

Consider making the caches `ThreadLocal`

Reusing objects: object pools

Classes become mutable

Instead of allocating, you take from the pool, then release

Preallocation: optional, but can make life easier for GC

Caveats:

- Memory leaks, you are responsible for lifecycle management
- If your objects point to other objects, clear them (better avoid)

Reusing objects: “stashes”

A special case of an object pool

Objects that only live during processing a request

A thread processes the request start-to-finish: one stash per thread

You can avoid lifecycle management: reclaim all after processing the request

“stashes”

```
public void run() {
    MyObject o1 = stash.retrieve()
    doWork(o1);
    MyObject o2 = stash.retrieve()
    doWork(o2, o1);
    ...
    stash.release()
}
```

Caveats, you’re responsible for:

- Not leaking the objects outside of the request
- Clearing objects if they point to other data

Off heap

Native `ByteBuffer` or `MappedByteBuffer`

```
buffer = ByteBuffer.allocateDirect();
buffer.put(index, value);
```

`sun.misc.Unsafe`:

```
address = Unsafe.allocateMemory();
Unsafe.putInt(address, value);
```

(de)serialization

Purpose: RPC protocols / off-heap storage

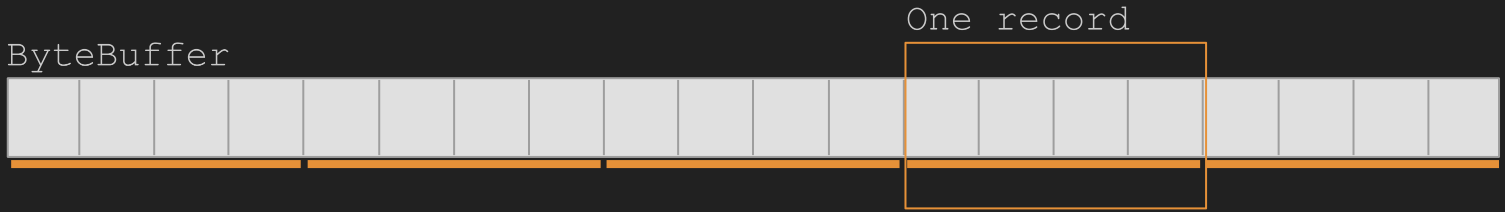
Simple: use a zero-allocation serialization library:

- SBE → github.com/real-logic/simple-binary-encoding/wiki
- FlatBuffers → google.github.io/flatbuffers/

They'll work on `ByteBuffer`s you provide:

- Can be off-heap based: `ByteBuffer.allocateDirect()`
- Pool them or use a larger `ByteBuffer` as a ring buffer

(de)serialization




```
ByteBuffer data = ...;
...
while (data.hasRemaining()) {
  ByteBuffer bytes = data.slice()
  bytes.limit(RECORD_SIZE)
  data.position(data.position() + RECORD_SIZE)
  users.add(new Record(bytes))
}
```

- Easy to build an `Iterator[Record]` over a `ByteBuffer`
- Single copy of the data (can also be memory-mapped)
- Easier to achieve cache friendliness

We’re hiring!

SCHIBSTED
MEDIA GROUP

Q&A

Thanks!