THE TROUBLE WITH
MEMORY
OUR MARKETING SLIDE

- Kirk Pepperdine
- Author of jPDM, a performance diagnostic model
- Author of the original Java Performance Tuning workshop
- Co-founded jClarity
  - Building the smart generation of performance diagnostic tooling
    - Bring predictability into the diagnostic process
- Co-founded JCrete
- The hottest unconference on the planet
- Java Champion(s)
What is your performance trouble spot
INDUSTRY SURVEY

What are the typical root causes you most often experience:

- 54.80% Slow database queries
- 38.60% Too many database queries
- 28.36% Concurrency issues
- 27.59% Memory leak
- 18.50% Slow DB
- 13.32% Excessive disk IO
- 12.87% Excessive network IO
- 11.97% Slow/unreliable third party entities
- 10.56% Excessive memory churn
- 7.55% HTTP session bloat
- 4.35% Don't know
- 1.73% Other
- 23.43% Configuration issues
- 17.93% GC pauses

Copyright 2019 Kirk Pepperdine
> 70% of all applications are bottlenecked on memory
and no, Garbage Collection is not a fault!!!!
DO YOU USE

Spring Boot

Copyright 2019 Kirk Pepperdine
DO YOU USE

Cassandra
or any big nosql solution
DO YOU USE Apache Spark
DO YOU USE

Apache Spark
or any big data framework
DO YOU USE

Log4J
DO YOU USE

Log4J
or any Java logging framework
DO YOU USE

JSON
DO YOU USE

JSON
With almost any Marshalling protocol
DO YOU USE

ECom caching products
DO YOU USE

ECom caching products
Hibernate

Copyright 2019 Kirk Pepperdine
DO YOU USE

ECom caching products
Hibernate
and so on
DO YOU USE

ECom caching products
Hibernate
and so on
and so on

Copyright 2019 Kirk Pepperdine
DO YOU USE

ECom caching products
Hibernate
and so on
and so on
and so on
and so on
then you are very likely in this 70%
WAR STORIES

» Reduced allocation rates from 1.8gb/sec to 0
  » tps jumped from 400,000 to 25,000,000!!!

» Stripped all logging out of a transactional engine
  » Throughput jumped by a factor of 4x

» Wrapped 2 logging statements in a web socket framework
  » Memory churn reduced by a factor of 2

» and .....
ALLOCATION SITE

```
Foo foo = new Foo();
```

forms an allocation site

```
0: new           #2   // class java/lang/Object
2: dup
4: invokespecial #1   // Method java/lang/Object."<init>";()V
```

- Allocation will (mostly) occur in Java heap
  - fast path
  - slow path
  - small objects maybe optimized to an on-stack allocation
Java Heap is made of:
- Eden - nursery
- Survivor - intermediate pool designed to delay promotion
- Tenured - to hold long lived data

Each space contributes to a different set of problems
- All affect GC overhead
EDEN ALLOCATIONS

top of heap pointer
OBJECT ALLOCATION

Foo foo = new Foo();
Bar bar = new Bar();
byte[] array = new byte[N];

top of heap pointer
FOO

Foo foo = new Foo();
Bar bar = new Bar();
byte[] array = new byte[N];
Foo foo = new Foo();
Bar bar = new Bar();
byte[] array = new byte[N];
Foo bar byte[]

Foo foo = new Foo();
Bar bar = new Bar();
byte[] array = new byte[N];
In multi-threaded apps, top of heap pointer must be surrounded by barriers

- single threads allocation
- hot memory address

solved by stripping (Thread local allocation blocks)
TLAB ALLOCATION

- Assume 2 threads
- each thread will have it’s own (set of) TLAB(s)
Thread 1 -> Foo foo = new Foo(); byte[] array = new byte[N];
- byte[] doesn’t fit in a TLAB
Thread 2 -> Bar bar = new Bar();
Threshold defining when to request a new TLAB
- prevent buffer overflows
- waste up to 1% of a TLAB
- Allocation failure to prevent buffer overflow
- somewhat expensive failure path
Allocations in tenured make use of a free list
- free list allocation is ~10x the cost of bump and run
- Data in tenured tends to be long lived
- amount of data in tenured do affect GC pause times
PROBLEMS

- High memory churn rates
- many temporary objects
PROBLEMS

- High memory churn rates
  - many temporary objects
- Quickly fill Eden
  - frequent young gc cycles
  - speeds up aging
  - premature promotion
  - more frequent tenured cycles
  - increased copy costs
  - increased heap fragmentation
- Allocation is quick
  - quick * large number = slow
REDUCING ALLOCATIONS

size of gain

> 1gb/sec

< 300mb/sec
PROBLEMS

- High memory churn rates
  - many temporary objects
- Large live data set size
  - inflated live data set size
  - loitering
PROBLEMS

- High memory churn rates
  - many temporary objects
- Large live data set size
  - inflated live data set size
  - loitering
- inflated scan for root times
- reduced page locality
- Inflated compaction times
  - increase copy costs
  - likely less space to copy too
PAUSE VS OCCUPANCY

Heap Occupancy After GC

GC Pause Time

GC Pause Time Over Time

Copyright 2019 Kirk Pepperdine
PROBLEMS

- High memory churn rates
  - many temporary objects

- Large live data set size
  - inflated live data set size
    - loitering

- Unstable live data set size
  - memory leak
PROBLEMS

- High memory churn rates
  - many temporary objects

- Large live data set size
  - inflated live data set size
    - loitering

- Unstable live data set size
  - memory leak

- Eventually you run out of heap space
  - each app thread throws an OutOfMemoryError and terminates
  - JVM will shutdown with all non-daemon threads terminate
Escape Analysis
Demo time
Ask me about our Java Performance Tuning Workshops