

# Scaling the OpenJDK



Claes Redestad  
Java SE Performance Team  
Oracle

JavaYourNext

(Cloud)

# Safe Harbor Statement

The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.

# In this talk...

We'll mainly concern ourselves with the **vertical** scalability of Java running on a single OpenJDK 9 JVM

For examples, I will be using JMH to explore code and bottlenecks

- <http://openjdk.java.net/projects/code-tools/jmh/>
- Benchmarking machine is a dual-socket Intel Xeon E5-2630 v3 (Haswell), 8 cores with 2 threads each

Don't take anything presented here to be good, general performance advice or even representative of what you'd see on your own hardware.

# JVM Scalability challenges

- Allow many concurrent and parallel tasks
- Allow for increasing memory requirements of applications
- Make it easy to work with
- Do all this without degrading throughput, latency and memory overheads (too much)!



Scaling up gently

JavaYourNext

(Cloud)

# Bottlenecks hiding in plain sight...

```
import org.openjdk.jmh.annotations.*;
import java.util.*;

@State(Scope.Benchmark)
public class Scale {
    public int year = 2017;
    public int month = 11;
    public int day = 29;

    @Benchmark
    public Date getDate() {
        return new Date(year, month, day);
    }
}
```

# Bottlenecks hiding in plain sight..

Benchmark	Mode	Cnt	Score	Error	Units			
Scale.getDate	avgt	10	0.637	± 0.027	us/op	#	1	thread
Scale.getDate	avgt	10	1.239	± 0.149	us/op	#	2	threads
Scale.getDate	avgt	10	9.713	± 0.676	us/op	#	8	threads
Scale.getDate	avgt	10	18.215	± 2.578	us/op	#	16	threads
Scale.getDate	avgt	10	37.693	± 2.374	us/op	#	32	threads

No scaling at all!

**Reason:** Date(int, int, int) synchronizes on a shared, mutable calendar instance!

# Better alternatives exist

```
public LocalDate getLocalDate() {  
    return LocalDate.of(year, month, day);  
}
```

Benchmark	Mode	Cnt	Score	Error	Units		
Scale.getLocalDate	avgt	10	0.031	± 0.007	us/op	#	1
Scale.getLocalDate	avgt	10	0.024	± 0.009	us/op	#	2
Scale.getLocalDate	avgt	10	0.029	± 0.005	us/op	#	8
Scale.getLocalDate	avgt	10	0.037	± 0.007	us/op	#	16
Scale.getLocalDate	avgt	10	0.067	± 0.001	us/op	#	32

Much better! Only a small overhead per operation when saturating all hyperthreads.



# Sharing effects

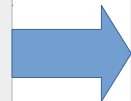
JavaYourNext

(Cloud)

# Improve String.hashCode implementation

During JDK 9 development, an innocent little clean-up was done to the String.hashCode implementation:

```
int h = hash;
if (h == 0 && value.length > 0) {
    char val[] = value;
    for (int i = 0;
        i < value.length;
        i++) {
        h = 31 * h + val[i];
    }
    hash = h;
}
```




```
int h = hash;
if (h == 0) {
    for (int v : value) {
        h = 31 * h + v;
    }
    hash = h;
}
```

# Improve String.hashCode a bit further...


For the corner case of the empty String we were now always calculating and storing 0 to the hash field.

Even though the value doesn't change, this causes the cache line to be evicted, which led to a 5% regression on a standard benchmark on dual-socket machines

```
int h = hash;
if (h == 0 && value.length > 0) {
    char val[] = value;
    for (int i = 0;
        i < value.length;
        i++) {
        h = 31 * h + val[i];
    }
    hash = h;
}
```



```
int h = hash;
if (h == 0) {
    for (int v : value) {
        h = 31 * h + v;
    }
    hash = h;
}
```



```
int h = hash;
if (h == 0) {
    for (int v : value) {
        h = 31 * h + v;
    }
    if (h != 0) {
        hash = h;
    }
}
```

# True sharing in "".hashCode()

Using JMHs perfnorm profiler made it easy to see not only the extra store in the second implemenation, but also the dramatic increase in L1 cache misses per operation induced by these stores:

```
$ java -jar benchmarks.jar .*String.hashCode.* -t 4 -prof perfnorm
```

String.hashCode2	avgt	5	38.701 ± 0.794	ns/op
String.hashCode2:CPI	avgt		3.501	#/op
String.hashCode2:L1-dcache-load-misses	avgt		<b>0.460</b>	#/op
String.hashCode2:L1-dcache-loads	avgt		14.173	#/op
String.hashCode2:L1-dcache-stores	avgt		<b>5.067</b>	#/op
String.hashCode3	avgt	5	6.512 ± 0.450	ns/op
String.hashCode3:CPI	avgt		0.527	#/op
String.hashCode3:L1-dcache-load-misses	avgt		<b>0.001</b>	#/op
String.hashCode3:L1-dcache-loads	avgt		13.995	#/op
String.hashCode3:L1-dcache-stores	avgt		<b>4.005</b>	#/op

(Omitted the first implementation as those results are indistinguishable from the third)

# Sharing another story...

- Around 2012 we got a new batch of multi-socket hardware where we started seeing intermittent performance regressions on various benchmarks not seen before
  - Often reductions of around 5-10% from one build to another, then back to normal after a build or two...
  - Then discovered that within the same build - with everything else exactly the same - performance could flip back and forth between a good and bad state by simply adding any parameter to the command line...
- It took a few false starts, but soon a possible root cause was found

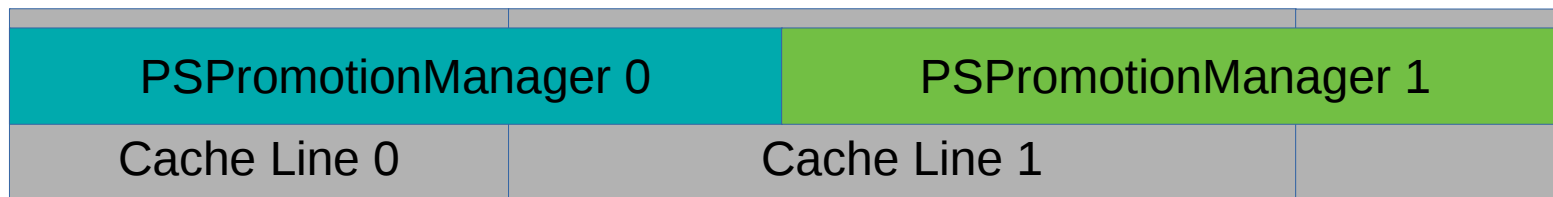
# The dreaded false sharing

- HotSpots parallel GC implementation has a PSPromotionManager class to keep information pertaining to an individual GC thread
- Each thread's instance of the PSPromotionManager is allocated at the same time and laid out next to each other in an array
- When aligned to cache lines, all was good

PSPromotionManager 0	PSPromotionManager 1
Cache Line 0	Cache Line 1

# The dreaded false sharing

- When memory layout changed, say, due to the addition of a command line flag, the alignment of the PSPromotionManager might shift so that they were now split across cache lines:

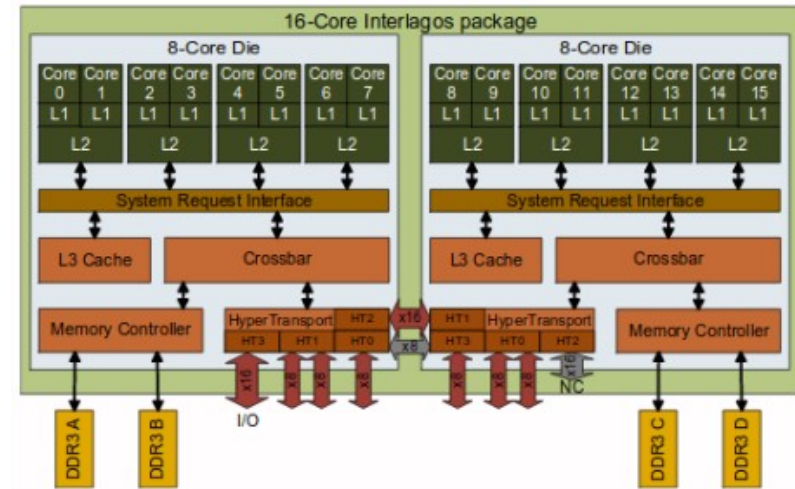


- When mutating state at the end of the first manager instance, memory pertaining to the second manager is dirtied and evicted from CPU caches

# False sharing explained

- On most multi-core CPU architectures, data stores will enable a cache coherency protocol to ensure the view of the memory is kept consistent across CPUs - atomicity guarantees may incur added cost
- False sharing happens when one (or more) CPUs write to memory that just happens to be on the same cache line as other memory that another CPU is working on

The cost of false sharing grows dramatically when work is spread across different sockets





# PSPromotionManager false sharing

Solution:

- Pad the PSPromotionManager object to be size aligned with cache line size
- Align the array of PSPromotionManager objects to start at a cache line boundary

Much more consistent performance in benchmarks using ParallelGC ever since!

Most code isn't as massively parallel as the stop-the-world parallel GC algorithm at play here, so it's not unlikely there are problems lurking elsewhere that are simply harder to detect or provoke...

# Contention everywhere!

- JEP 143: Improved Contended Locking was a project delivered as part of JDK 9
  - Java differentiates between *biased locks*, *thin locks*, and heavy, contended *monitors*
  - Biased or thin locks are used as a fast-path when application code needs to lock on an Object but noone else is contending for the lock
  - Monitors are installed, or *inflated*, into Objects when demand for the Object monitor becomes contentious
- The JEP work includes a number of small but well-rounded optimizations to reduce the overall latencies of synchronization primitives in java

# Pad all the things!

- One optimization of JEP-143 was to pad the native data structure used to represent the contended monitor of an Object
  - Up to 50% improvements in *targetted* microbenchmarks
- Global monitors and mutexes in HotSpot were later padded out, too
- Similarly there is the `@jdk.internal.vm.annotation.Contended` facility (since JDK 8) to apply padding around a Java field and objects

# Inherent VM latencies

JavaYourNext

(Cloud)

# Safepoints, GC and VM operations

- When the VM for some reason needs to stop all threads, it requests a safepoint that instructs all application threads to stop
- Java threads perform safepoint polls at regular intervals during normal execution - if a safepoint has been requested the thread will halt and not resume work until the safepoint has completed
- Safepoints can be initiated by the GC or the VM, and by default the VM will safepoint at least once per second
- While typically small and quick, time spent in safepoints and GC do put upper bounds on scalability

# Monitor deflation and other cleanup...

- One VM operation performed at safepoints is scanning for Java object monitors to *deflate*, which means removing the object monitor from an Object and recycling it to a global list.
- A single thread scans the entire population of monitors...
- ... and both in benchmarks and real world applications, the monitor population might grow to 100s of thousands, causing this deflation operation to take significant time
- JDK 9: Made -XX:+MonitorInUseLists default (+ deprecated the flag)
- Future work:
  - Concurrent Monitor Deflation
  - Parallelize safepoint cleanup

# JEP 312: Thread-local handshakes (JDK 10)

- Added infrastructure to move from only allowing global safepoints (that stop all threads) to perform handshakes on any number of threads.

"The big difference between safepointing and handshaking is that the per thread operation will be performed on all threads as soon as possible and they will continue to execute as soon as its own operation is completed."

- **Enables** a number of optimizations, such as no longer needing to stop all threads to revoke a biased lock

# UseMembar - Not all optimizations age well

- -XX:-UseMembar was implemented at a time when fence instructions on the state of the art hardware were expensive. Instead a "pseudo-membar" was introduced.
- Turns out this implementation caused various issues, including a scalability bottleneck when reading thread states, a long tail of actual bugs... there's even some false sharing possible when running more than ~64 java threads
- Making +UseMembar the default trades global synchronization of thread state for local fences
  - In single-threaded benchmarks that take a lot of transitions this can be a performance loss since fences still have higher latencies
  - For scalability, however, it's typically preferable to take a local performance hit if it removes a cost incurred on all threads



GC scalability

JavaYourNext

(Cloud)

# G1 scalability improvements in JDK 9

- 10 TB dataset on :
  - "Pause times reduced by 5-20x on read-only benchmark"
  - "For the first time we achieved stable operation on a mixed read-write workload with a 10 TB dataset"

<https://www.youtube.com/watch?v=LppgqvKOUKs>
- Key improvements includes merging per-thread bitmaps into a single shared structure managed by lock-free algorithms, dropping worst case mark times from 15 minutes to mere seconds mainly from becoming way more cache-friendly.

# ZGC project proposed

- Max pause times not exceeding 10ms on multi-TB heaps
- Parallel *and* concurrent: Designed for 1000s of hardware threads
  - => Requires lock- and/or wait-free datastructures
  - => Thread/CPU local data
- Dynamic, NUMA-aware and lazy resource allocation
- Features striping: tries to be locality aware and allocate into compact chunks of memory that individual GC workers tend to, aiming to reduce memory contention and cache cross-talk.
- Sacrifice a bit of throughput to improve latency and scalability

# Honorable mention: Project Loom

JavaYourNext

(Cloud)

# Project Loom

- Implement a light-weight user-mode thread as an alternative to the OS-managed threads that are the de facto standard in Java today
- The goal is that such "threads" will:
  - Have minimal footprint - allow millions of concurrent jobs on a system which could host only thousands of Threads
  - Remove penalties of blocking
  - Support tail calls...
- 5 minute lightning talk:  
<https://www.youtube.com/watch?v=T-8fA3dEUlg>

# Q & A