KODEWERK

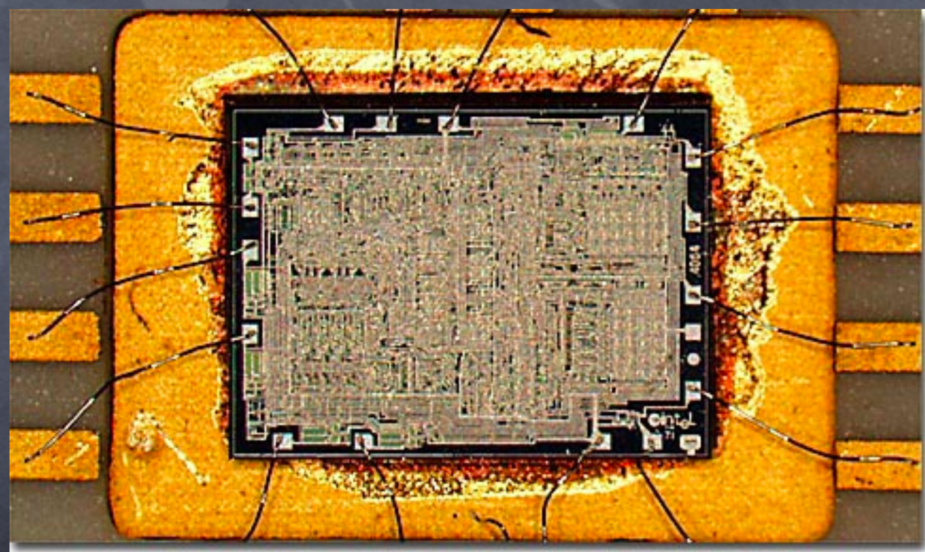
Java 8
New or Noteworthy!

KODEWERK

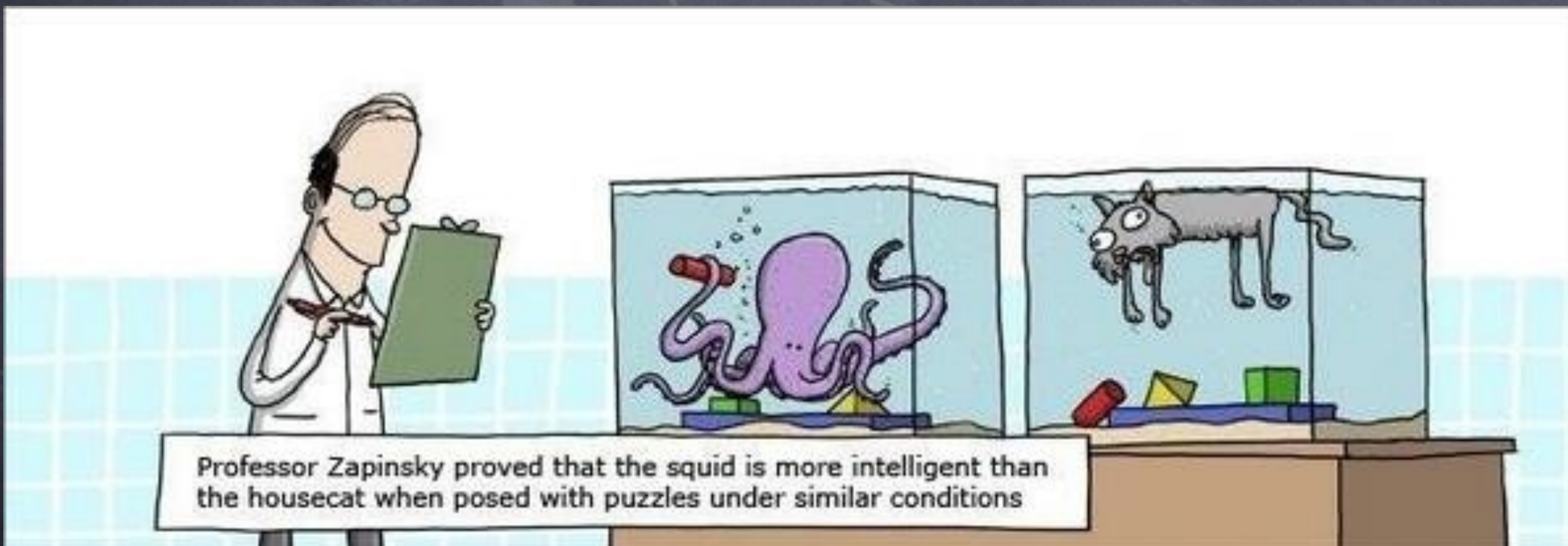
About Me

- Founder and CTO of jClarity
  - next gen performance diagnostic engine
- Performance tuning and training
- Helped establish www.javaperformancetuning.com
- Member of Java Champion program
- Other stuff... (google is you care to)

What is this?



Benchmark Alert!!!!!!



The Big News in Java 8

λ Expressions

```
LambdaParameters '->' LambdaBody

() -> 42
(x,y) -> x * y
(int x, int y) -> x * y
```

A Logging Surprise

- Logging with Lambda can offer some advantages
  - lazy execution

```
for ( int i = 0; i < 5000; i++) {
    LOGGER.fine ("Trace value: " +
getValue());
}
```

3200ms

```
for ( int i = 0; i < 5000; i++) {
    LOGGER.fine (() -> "Trace value: " + getValue());
}
```

35ms

```
for ( int i = 0; i < 5000; i++) {
    if ( LOGGER.isLoggable( Level.FINE))
        LOGGER.fine ("Trace value: " + getValue());
}
```

0ms

What about the other stuff?

- Including Lambda's, there are 55 JEPs assigned to Java 8
  - Java Enhancement Proposal
    - Support enhancements to the JDK
  - JEP 1 describes the process (http://openjdk.java.net/jeps/1)
    - JEP 2.0 is being worked on.
- Many other smaller features or changes not mentioned in a JEP
  - patches to existing code and small features added

Nashorn

- JavaScript in the JVM
- First use case for adding direct support for dynamic languages in the JVM
  - many lessons learned in addition to those learned from other dynamic languages
    - JRuby, Clojure.....
- Java calls JavaScript
- JavaScript calls Java

Tiered Compilation

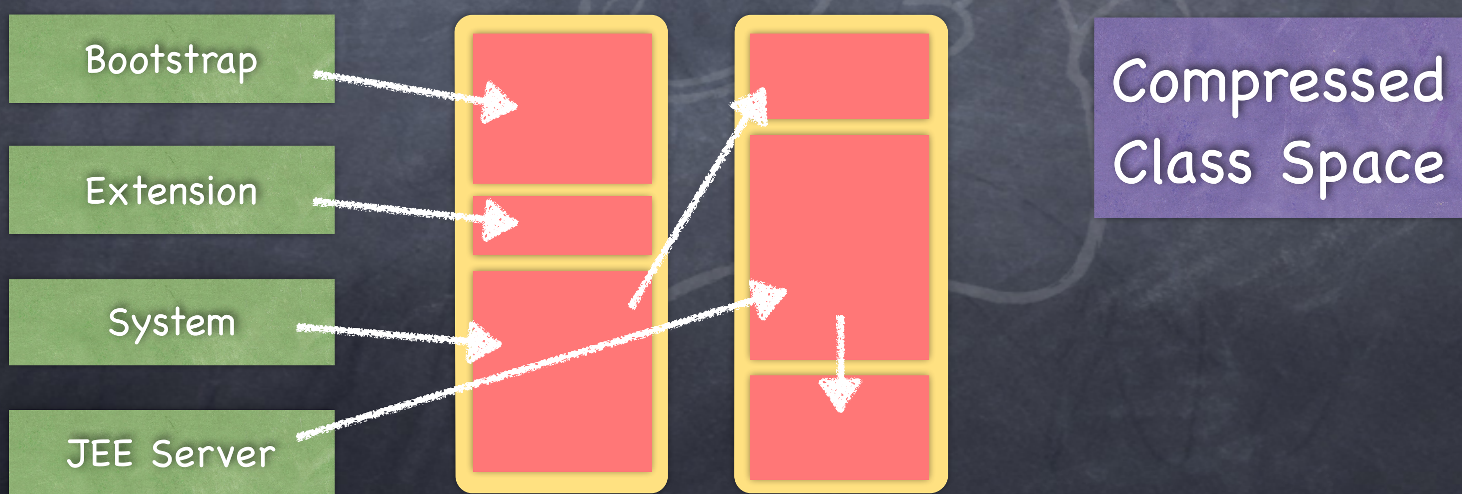
- C1 or Client HotSpot
- C2 or Server or Optimizing HotSpot
- Tiered combines the fast C1 with the deeper C2 optimizations
  - default in 1.8.0
  - may require a bigger code cache
  - currently not quite as stable as C1 or C2

Permspace Removal

- Permspace is a fixed size memory pool for things that should never be GC'ed
    - class meta data, vtable
    - method meta data
    - Interned String table (moved to Java heap in 1.7.0_40)
    - constant pool
- Permspace leaks due to class relationships and references to/from Classloaders

Metaspace

- Metaspace is a C heap data structure designed to hold class meta data
  - classloader allocates one or more meta chunks in one or more virtual spaces in the



KODEWERK

Metaspace Maintenance

- Meta chunks returned to free list when classloaer is GC'ed
  - not scanned by GC
  - no individual reclamation
- Virtual memory spaces returned when emptied
- -XX:MetaspaceSize=<N[G,M,K,B]>
  - sets a high water mark for a Metaspace GC
- -XX:MaxMetaspaceSize=<N>
  - size of Metaspace is otherwise unbounded

Metaspace Maintenance

- Pointers to Class meta data are compressed by default
  - UseCompressedClassPointers
- Filling CompressedClassSpace can result in an OOME
  - CompressedClassSpaceSize
  - Max size is 4G

Metaspace Tooling

- MemoryManagerMXBean::MetaspaceManager
- MemoryPoolMXBeans Metaspace and Compressed Class Space
- jmap -clstats <pid>
- jstat -gc <pid> or jstat -class <pid>
- jcmd <pid> GC.class_stats
  - need to set UnlockDiagnosticVMOptions
- jcmd <pid> VM.native_memory
  - need to turn on native memory tracking (performance hit)

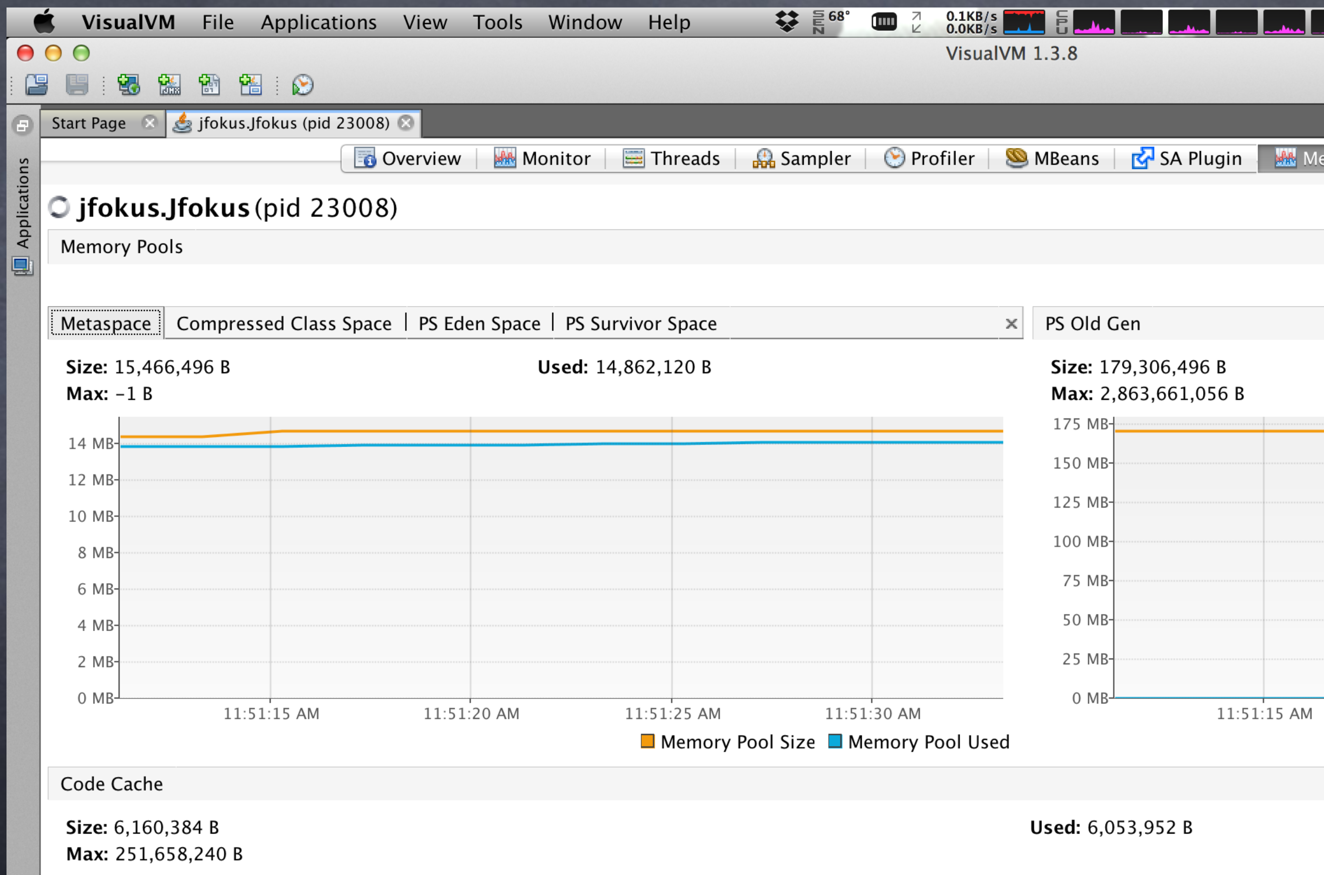
jmap -clstat

```
class_loader classes bytes parent_loader alive? type

<bootstrap> 836 1580696   null   live <internal>
0x000000076ab58ba0 18 49113 0x000000076ab3b7d8 live sun/misc/Launcher$AppClassLoader@0x00000007c0038320
0x000000076b250a00 7 31196 0x000000076ab3b7d8 live jdk/nashorn/internal/runtime/
StructureLoader@0x00000007c0067908
0x000000076bff1c28 15 91015 0x000000076ab58ba0 live jdk/nashorn/internal/runtime/
ScriptLoader@0x00000007c012c060
0x000000076ab3b7d8 770 1638365   null   live sun/misc/Launcher$ExtClassLoader@0x00000007c002d338
0x000000076c6b3950 12 62501 0x000000076ab58ba0 dead jdk/nashorn/internal/runtime/
ScriptLoader@0x00000007c012c060
0x000000076afc9ed0 0 0 0x000000076ab58ba0 live java/util/ResourceBundle
$RBClassLoader@0x00000007c0071548
0x000000076c2c8d60 10 51073 0x000000076ab58ba0 dead jdk/nashorn/internal/runtime/
ScriptLoader@0x00000007c012c060

total = 8 1668 3503959     N/A     alive=6, dead=2     N/A
```

VisualVM MemoryPoolView



Take aways

- Class metadata is being managed differently
  - will have to reconsider some tuning options
- You won't see OOME Permspace anymore
  - you can still have classloader leaks and they will be harder to find
    - leak will appear in C heap (process size gets bigger and bigger)

Retired GC Combinations

- Normal combinations
  - ParNew + CMS
  - DefNew + Serial Old
- Retired
  - DefNew + CMS
  - ParNew + Serial Old
  - Incremental CMS
- G1GC works as of 1.8.0_20, 1.7.0_51

HashMap Collision Handling

- Collisions are handled in a balanced tree
  - were handled with a linked list
  - worst case moves from O(n) to log(N).
- Linked list is reversed when resizing a map
  - thread performing an unsychronized reads during a resize can be trapped in an infinite loop
  - symptom: burn a CPU

No matter how hard we’ve tried,
we’ve not been able to reproduce this
with the balance tree implementation

This does not mean you can safely use Hashmap
concurrently without synchronization

Stream

- Defines an internal iterator over a collections
- Stream operations are categorized as an intermediate or a terminator
  - intermediate operations produce a stream
    - filter with lazy evaluation of Predicates
    - map to intermediate values
  - terminal operators produce values

Stream Execution

- Combined to form a stream pipeline
  - data source -> intermediate -> intermediate -> terminating
- Processing starts when you hit a terminator
- Easily paralellized
  - supported internally by providing a Splitorator
    - an Iterator that knows how to decompose the stream into sub-streams

Streams

- Defined in interface Collection::stream()
- Many other classes implement stream()
  - Arrays.stream(Object[]),
  - Stream::of(Object[]), ::iterate(Object,UnaryOperator)
  - File.lines(), BufferedReader.lines(), Random.ints(), JarFile.stream()

Streams

```
gcLogEntries.
    stream().
    map(applicationStoppedTimePattern::matcher).
    filter(Matcher::find).
    mapToDouble(matcher -> Double.parseDouble(matcher.group(2))).
    summaryStatistics();
```

KODEWERK

Streams

data source

start streaming

map to Matcher

```
gcLogEntries.
    stream().
    map(applicationStoppedTimePattern::matcher).
    filter(Matcher::find).
    mapToDouble(matcher -> Double.parseDouble(matcher.group(2))).
    summaryStatistics();
```

filter out uninteresting bits

map group to Double

aggregate values in the stream

KODEWERK

ParallelStreams

```
gcLogEntries.
        parallelStream().
        map(applicationStoppedTimePattern::matcher).
        filter(Matcher::find).
        mapToDouble(matcher -> Double.parseDouble(matcher.group(2))).
        summaryStatistics();
```

data source

submit job Task to Fork-Join

map to Matcher

filter out uninteresting bits

map group to Double

aggregate values in the stream

Fork-Join

- Support for Jork-Join was put into JDK 7.0
  - difficult coding idiom to master
- Streams combined with Lambda's make this framework more reachable
  - how fork-join works and performs is important to your latency

Fork-Join

- Apply to a ParallelStream
  - break the stream up into chunks and submit each chunk as a ForkJoinTask
  - apply filter().map().reduce() to each ForkJoinTask
  - Call ForkJoinPool.get() to retrieve results

Fork-Join Performance

- Fork Join comes with significant overhead
  - each chunk of work must be large enough to amortize the overhead
  - C/P/N/Q performance model

    C - number of submitters
    P - number of CPUs
    N - number of elements
    Q - cost of the operation

C/P/N/Q

- Need to offset the overheads of setting up for parallelism
  - NQ needs to be large
    - Q can often only be estimated
    - N often should be > 10,000 elements
    - C may not be your limiting constraint

Kernel Times

- CPU will not be the limiting factor when
  - CPU is not saturated
  - kernel times exceed 10% of user time
- In this case adding **more** threads will make the situation worse!
  - predicted by Little's Law

Common Thread Pool

- Fork-Join by default uses a common thread pool
  - default number of worker threads == number of logical cores
- Performance is tied to which ever you run out of first
  - availability of the constraining resource
  - number of ForkJoinWorkerThreads

ForkJoinPool

```
public void parallel() throws IOException {

ForkJoinPool forkJoinPool = new ForkJoinPool(10);
String<String> stream = Files.lines(new File(gcLogFileName).toPath());
forkJoinPool.submit(() ->
            stream.parallel().
                    map(applicationStoppedTimePattern::matcher).
                    filter(Matcher::find).
                    mapToDouble(matcher -> Double.parseDouble(matcher.group(1))).
                    summaryStatistics().toString();
}
```

Little's Law

- Fork-Join is a work queue
  - work queue behavior is typically modeled using Little's Law
- States that number of task in a system equals the arrival rate times the amount of time it takes to clear an item
- Example: System has a requirement of 400 TPS. It takes 300ms to process a request

Number of tasks in system = 0.300 * 417 = 125

Components of Latency

- Latency is the time from stimulus to result
  - internally latency consists of active and dead time
- If (thread pool is set to 8 threads) and (task is not CPU bound)
  - task are sitting in queue accumulating dead time
  - make thread pool bigger to reduce dead time

From The Previous Example

125 tasks in system - 8 active = 117 collecting dead time

Conclusion:

```
if there is capacity to cope then
    make the pool bigger
else
    add capacity or tune to reduce strength of the dependency
```

KODEWERK

Instrumenting ForkJoinPool

- We can get the statistics needed from ForkJoinPool needed for Little's Law
  - need to instrument ForkJoinTask::invoke()

```
public final V invoke() {
    ForkJoinPool.common.getMonitor().submitTask(this);
    int s;
    if ((s = doInvoke() & DONE_MASK) != NORMAL)
        reportException(s);
    ForkJoinPool.common.getMonitor().retireTask(this);
    return getRawResult();
}
```

- Collect invocation interval and service time
  - code is in Adopt-OpenJDK github repository

Performance Implications

- In an environment where you have many parallelStream() operations all running concurrently performance maybe limited by the size of the common thread pool
- Can adjust the size of the default ForkJoinPool
  - -Dutil.concurrent.ForkJoinPool.common.parallelism=N
  - java.util.concurrent.ForkJoinPool.common.threadFactory
  - java.util.concurrent.ForkJoinPool.common.exceptionHandler
  - Runtime.getRuntime().availableProcessors();

Performance Implications

- Can submit to your own ForkJoinPool
  - must call get() on pool to retrieve results
  - beware: performance will be limited by the constraining resource
  - not an officially supported idiom

```
new ForkJoinPool(16).submit(() -> ……… ).get()
```

Performance Implications

Constraining Resource: I/O Logical Cores: 8 ThreadPool: 8	Tasks Submitted	Time in ForkJoinPool (seconds)	Inter-request Interval (seconds)	Expected Number of Tasks in ForkJoinPool	Total Run Time (seconds)
Lambda Parallel	20	2.5	2.5	1	50
Lambda Serial	0	6.1	0	0	123
Sequential Parallel	20	1.9	1.9	1	38
Concurrent Parallel	20	3.2	1.9	1.7	39
Concurrent Flood (FJ)	20	6.0	1.9	3.2	38
Concurrent Flood (stream)	0	2.1	0	0	41

KODEWERK

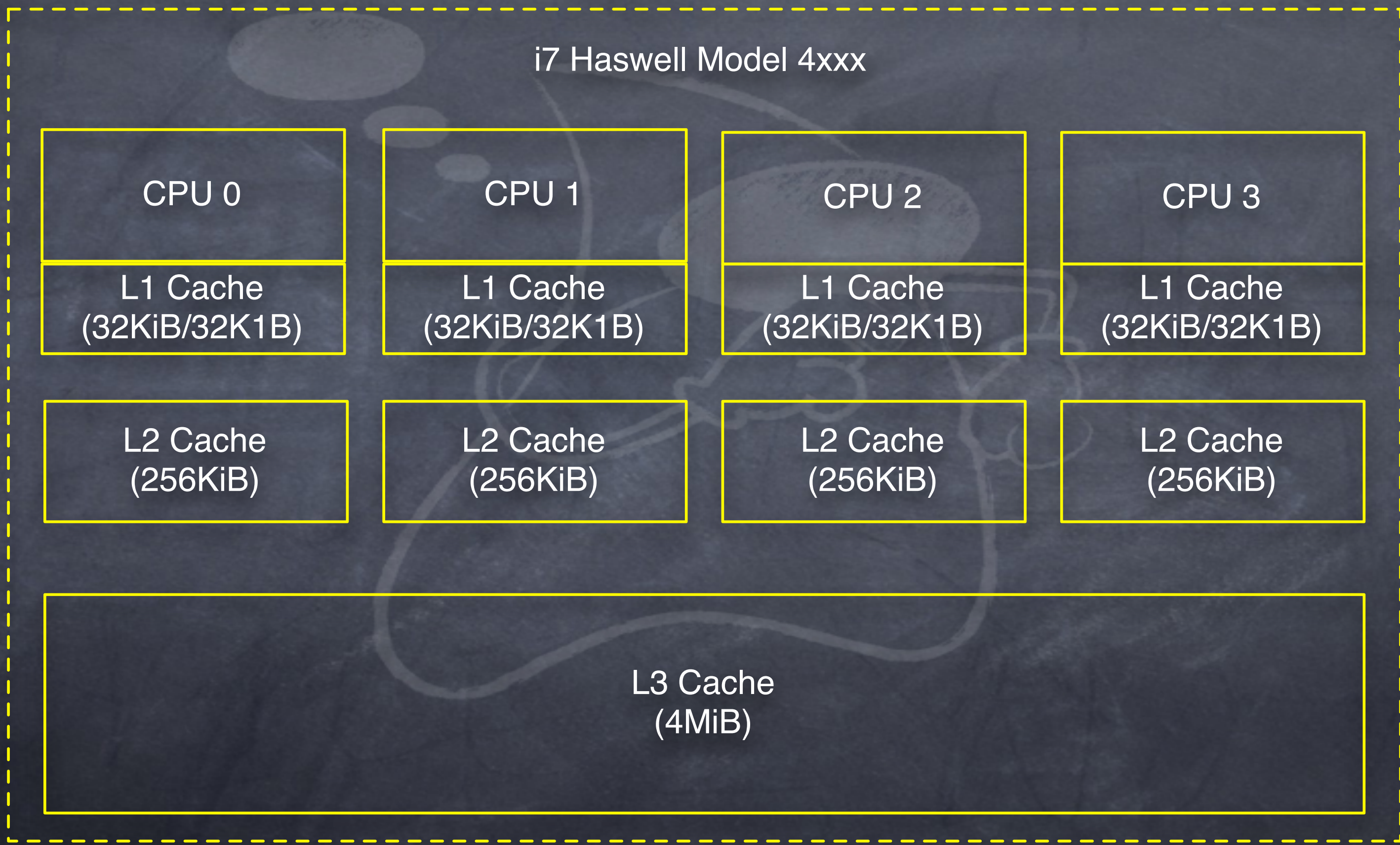
Performance Implications

Constraining Resource: CPU Logical Cores: 8 ThreadPool: 8	Tasks Submitted	Time in ForkJoinPool (seconds)	Inter-request Interval (seconds)	Expected Number of Tasks in ForkJoinPool	Total Run Time (seconds)
Lambda Parallel	20	2.8	2.8	1	56
Lambda Serial	0	7.5	0	0	150
Sequential Parallel	20	2.6	2.6	1	52
Concurrent Parallel	20	5.8	3.0	1.9	60
Concurrent Flood (FJ)	20	43	6.5	6.6	130
Concurrent Flood (stream)	0	3.0	0	0	61

Take Aways

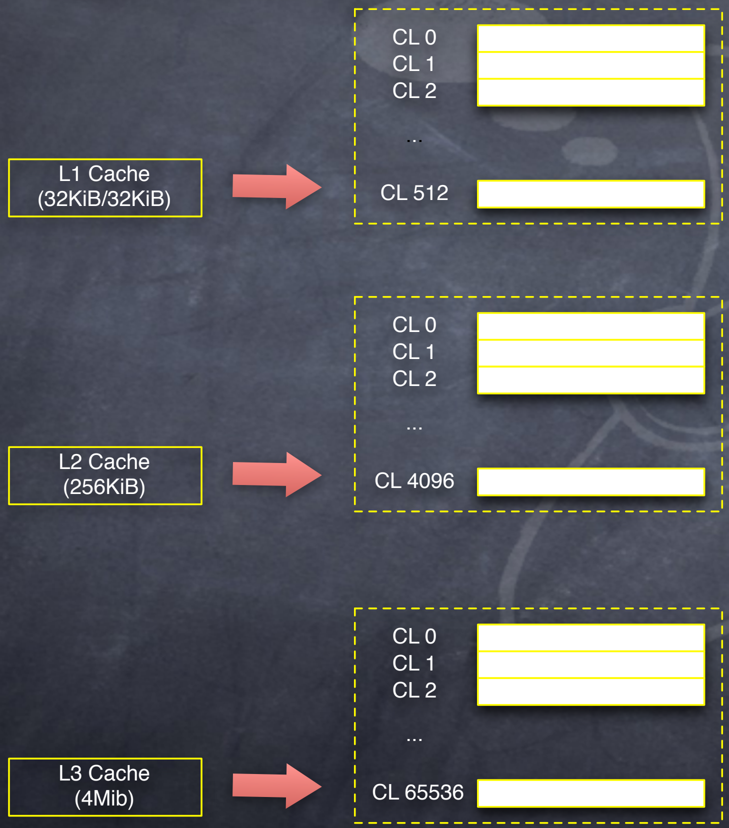
- Going parallel might not bring you the gains you expect
  - you may not know this until you hit production!
- Monitoring internals of JDK is important to understanding where bottlenecks are
  - JDK is not all that well instrumented
- You need to re-read the javadocs even for your old familiar classes
  - API's have changed to support streams

CPU Layout



i7 Haswell Model 4xxx
CPU 0
CPU 1
CPU 2
CPU 3
L1 Cache (32KiB/32K1B)
L1 Cache (32KiB/32K1B)
L1 Cache (32KiB/32K1B)
L1 Cache (32KiB/32K1B)
L2 Cache (256KiB)
L2 Cache (256KiB)
L2 Cache (256KiB)
L2 Cache (256KiB)
L3 Cache (4MiB)

CPU Caches and Cache Lines



- CPU caches are tables of cache lines
- Cache line is a fix size block of data
  - minimum chunk size data a CPU cache will work with
  - common size is 64 bytes
- CPU caches include
  - data, instruction, and TLB

MESI

- One of many memory models that describes how to treat cache lines
- Cache line can be in one of four states
  - Modified, Exclusive Shared, Invalid
- A cache line loaded into CPU 0's L1/L2 cache will be marked Exclusive
- If loaded into another CPU's L1/L2 cache, it will be marked shared

KODEWERK

Writes Are Expensive

- Before a CPU writes to a Shared cache line, it must first call for a RFO
    - read for ownership
- Before a CPU can write to an Exclusive cache lines if must first snoop all other reads
- Modified cache lines will be written to a store buffer
- Invalid cache lines must be refreshed
    - store buffers must be drained (fence)

Java and Cache Lines

- Several Java primitives or OOPs will fit into a single cache line
  - CPU's unit of atomicity > Java's unit of atomicity
- Java's classloader will reogranize and pack data

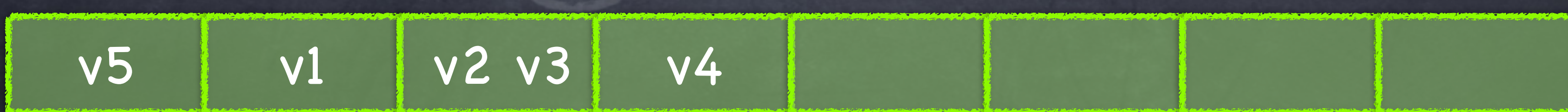
doubles	long	8 bytes
int	floats	4 bytes
shorts	char	2 bytes
booleans	byte	1 byte
references	4 or 8 bytes	
Repeat for subclasses		

Classloading

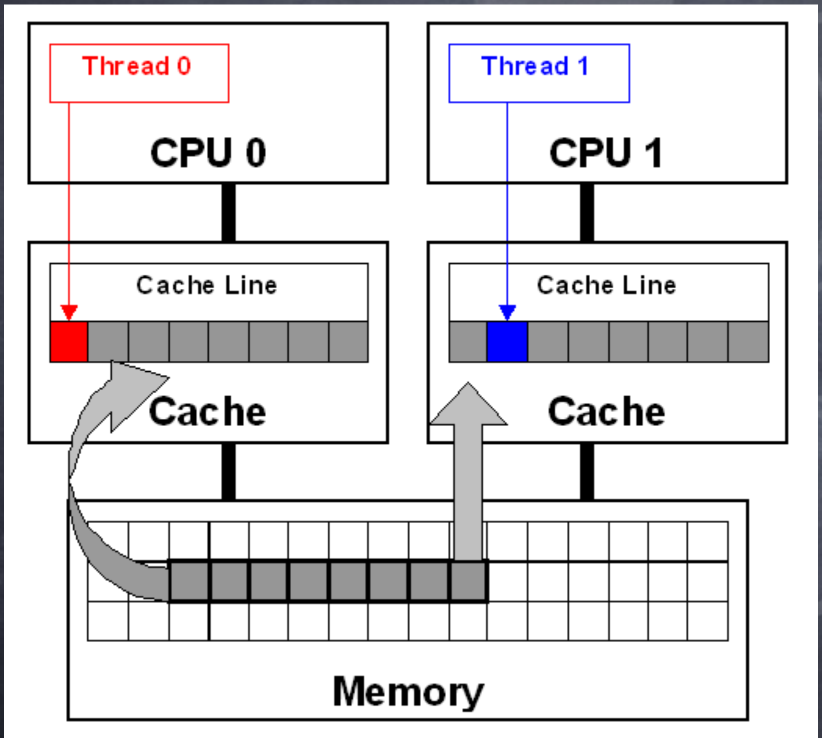
```
public class foo {
    private long v1;
    private int v2, v3,
    private Object v4;
    private double v5;
……
```

```
-XX:+PrintFieldLayout (debug build only)
@140 --- instance fields start ---
@140 "v5" D;
@148 "v1" J;
@156 "v2" I;
@160 "v3" I;
@164 "v4" Ljava.lang.Object;
@172 --- instance fields end ---
@172 --- instance ends ---
```

Cache line



False Sharing



- Two unrelated variables end up in the same cache line
  - thread 0 modifies one variable
  - thread 1 modified the other variable
- Each thread invalidates each others copy of the cache line
  - results in excessive numbers of cache misses, drop in retirement rates
- Diagnose using MSRs

Solution

- Arrange falsely shared variables so they don’t end up in the same cache line
  - place falsely shared variables into a subclass or superclass relationship
  - add padding in-between the variables
    - doesn’t work in Java 7 as DVE will JIT away the padding
  - @Contended annotion

@Contended

- Two flags involved
  - RestrictContended
    - default true
    - restricts @Contended annotation to JDK (trusted) classes
  - EnableContended
    - default true
    - enables @Contended annotation support

Using @Contended

- Suggests that a variable should be isolated from other variables

```
public class Point {
    int x;
    @Contended
    int y;
}
```

- Requries that annotations be operable on any Java type (new to Java 8)

Performance

- 1 thread not padded: 0.532 seconds
  - CPU consumption: 100%
- 1 thread padded: 0.522 seconds
  - CPU consumption: 100%
- 8 threads not padded: 8.31
  - CPU consumption: 800%
- 8 thread padded: 1.29
  - CPU consumption 800%

Take Aways

- False sharing is hard to detect
  - currently no reliable tooling
  - L2/L3 Cache hit/miss ratios and Instruction retirements rates
- Normally only part of a bottom up tuning regime
  - low latency
- Test affected code in isolation
- Need to adjust code to fit the hardware
  - runs coutner to "normal" thinking for Java developers

Things I Would Have Liked to Cover

- jdeps, a tool to discover your applications dependencies
- New Date and Time (based on JODA Time)
- A whole bunch of concurrency stuff
  - stamped lock, hires counters
  - updates to Unsafe (fences)
- And more……..


Java Performance Tuning,
May 26-29, Chania Greece
www.kodewerk.com