Java and the Machine

jClarity

http://www.jclarity.com

Our background

- **jClarity** - We use statistics and Machine Learning (ML) to find the root cause of performance problems

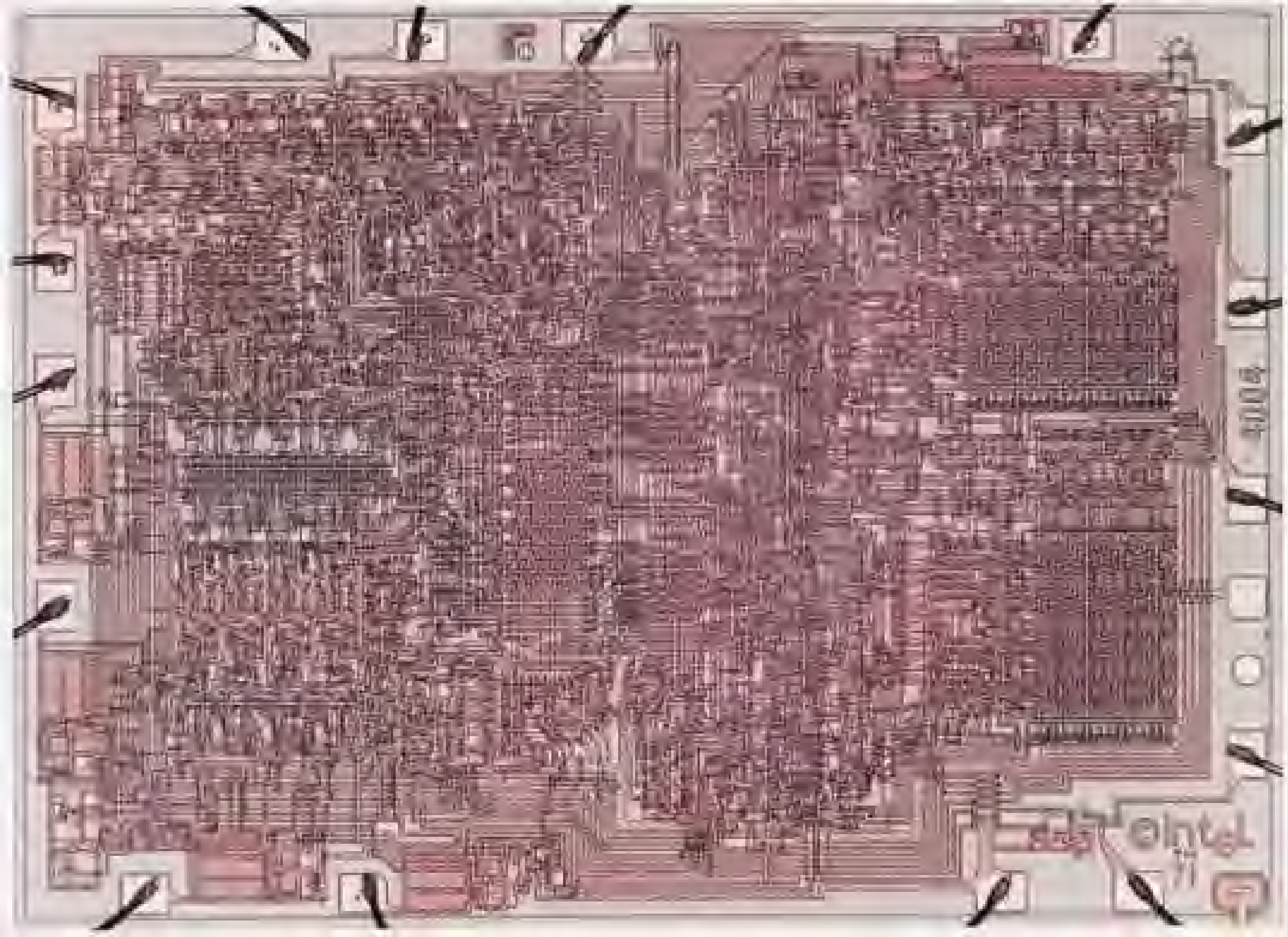
- **Martijn** - CEO & Janitor, Author, Speaker, Sun/Oracle Java Champion

Outline

1. Hardware has changed
2. Computer Science Laws (for performance)
3. Challenges with Java and the JVM
4. Java performance is hard to diagnose
5. Analytics > Metrics - an example with GC

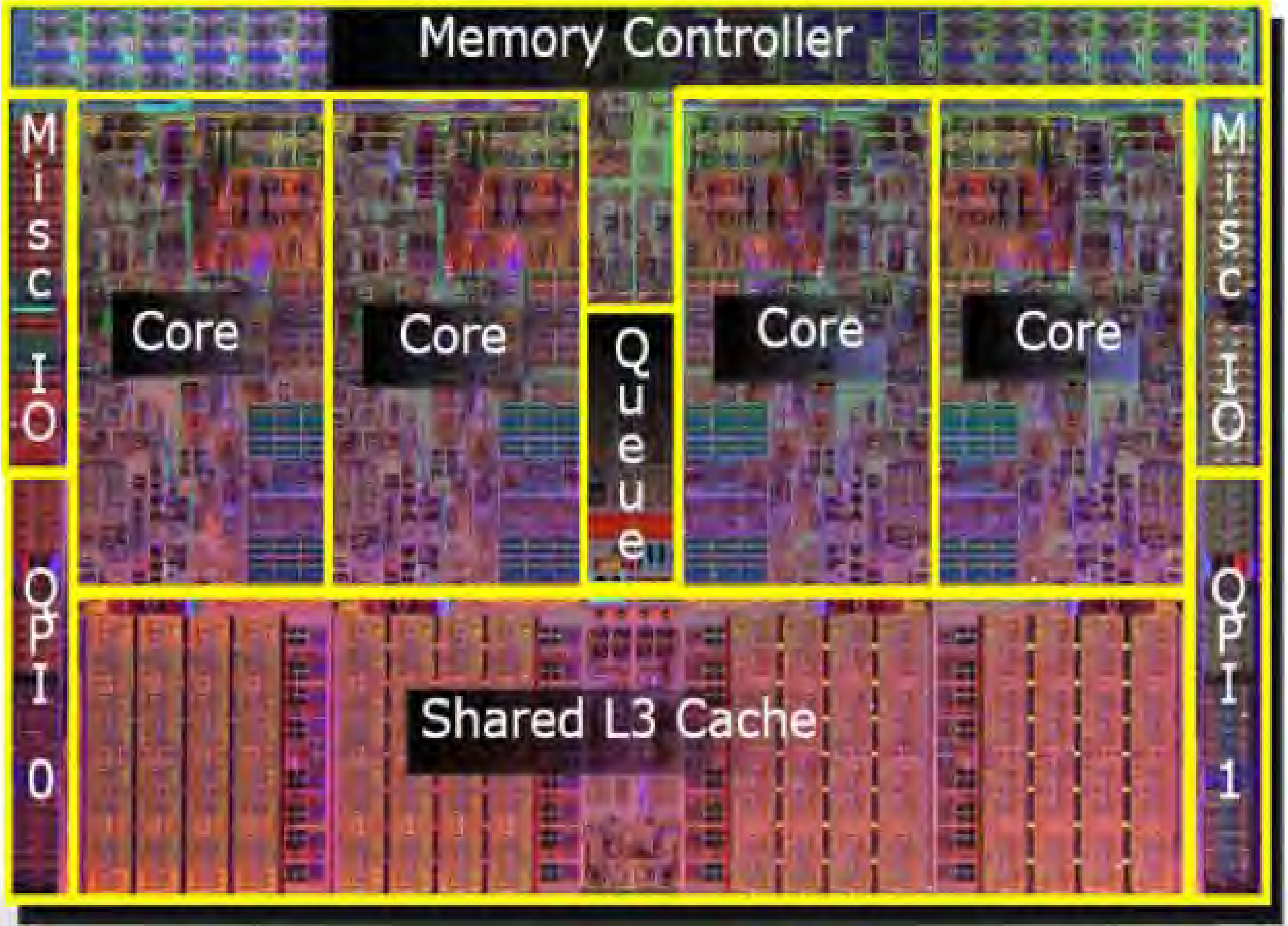
1. Hardware has changed

The next two slides show you an example of how hardware has changed



Intel 4004

The first commercial Microprocessor in 1971



Intel i7-3770

A more modern CPU

万事开头难

All things are difficult before they are easy

2. Computer Science Laws

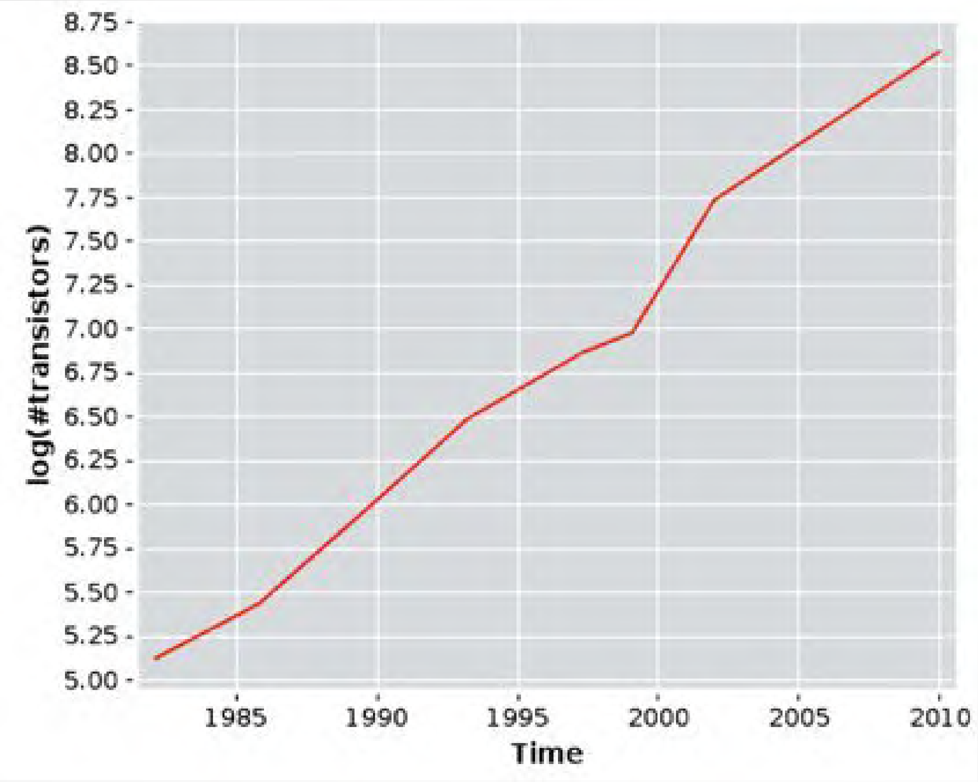
We are fortunate that we have good laws to understand this new world!

The 4 Performance Laws

The following 4 laws are important to understand for software performance

1. Moore's Law
2. Little's Law
3. Amdahl's Law
4. Gunter's Law

Moore's Law



The number of integrated circuits **double** every year

Little's Law

$$L = \lambda * W$$

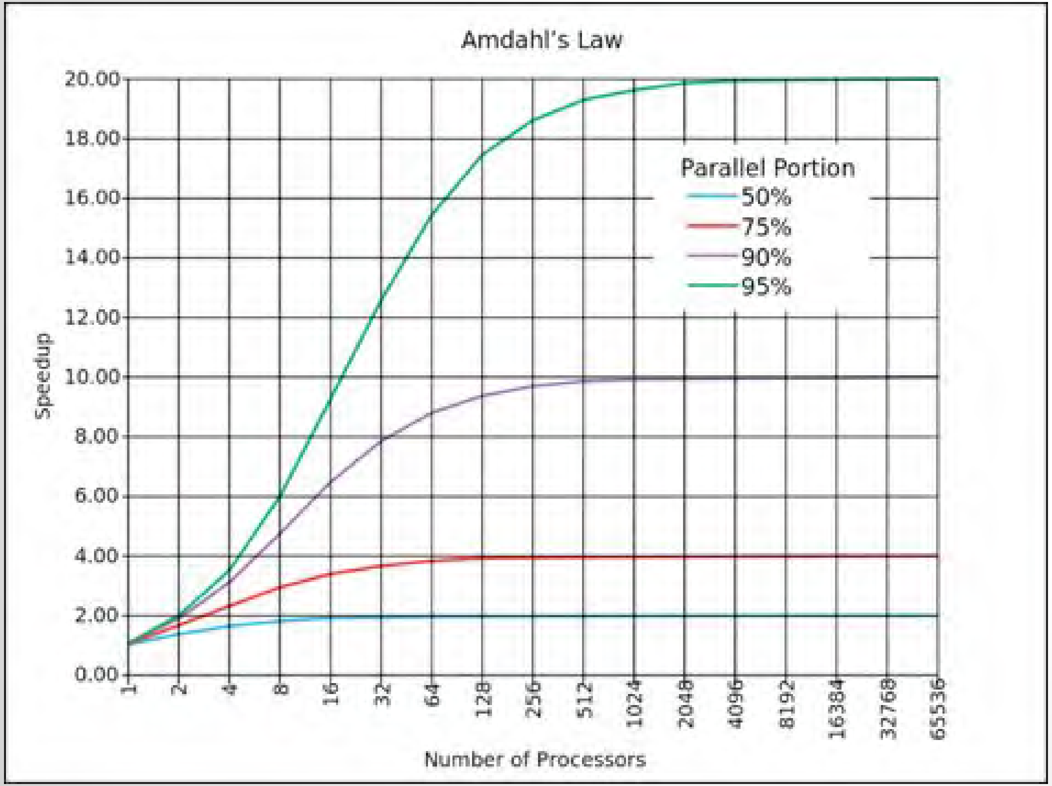
Throughput = Arrival Rate * Wait time

Little's Law - Example

$$\mathbf{500\ (L)} = 1000\ (\lambda) * 0.5\ (W)$$

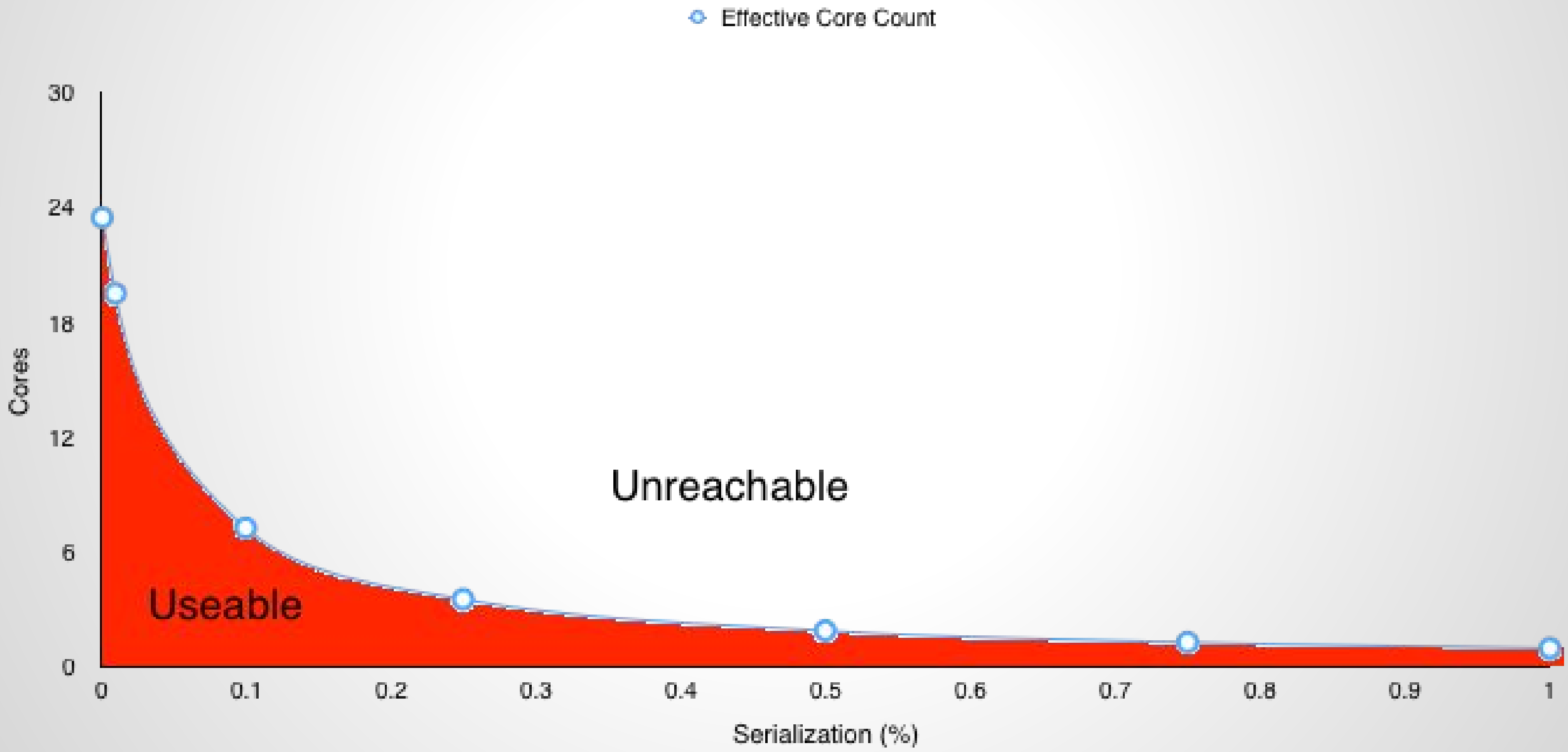
Average number of people =
arrivals per hour * length of stay

Amdahl's Law



$$\frac{1}{(1-P)+\frac{P}{S}}$$

Amdahl's Law - Inverse



Amdahl's Law - Examples

$$\frac{1}{(1-P)+\frac{P}{S}}$$

P = Percentage of algorithm that can be made faster
S = How much the algorithm can be sped up

Examples of P and S values

P = 0.3 30% of the algorithm can be made faster
S = 2 the algorithm can go twice as fast

Gunter's Law

Describes the relationship between Concurrency, Contention and Coherency

Coherency is the cost of the communication overhead between nodes

If coherency is 0, then Gunter's Law == Amdahl's law

3. Challenges with Java and the JVM

- Write Once Run Anywhere (WORA)
- Cost of the strong memory model
- Garbage Collection (GC) Scalability
- Container and Virtualisation support

Write Once Run Anywhere (WORA)

- CPU Differences
  - When are you allowed to cache or reorder?
- File System differences
  - O/S level support for symbolic links etc
- Display devices
  - Impossible to keep up with new hardware!

Write Once Run Anywhere (WORA)

- Native library support differences
  - Not all native libraries are equal!
- Operating System threading models
  - Threads are scheduled very differently
- No real GPU support

Cost of a strong memory model

- The JVM is very careful
  - Correctness > Performance!
- Locks enforce correctness
  - High cost to performance
- Locks define regions of serialization
  - Remember Little's law and Amdahl's law?

Garbage Collection (GC) scalability

- JVM traces live objects
  - Larger heaps usually means more objects
  - GC takes longer to find live objects
  - GC takes longer to manage heap during a collection
- No value types or structs in Java
  - Lots of inefficient object creation

Container & Virtualisation support

- Java does not access virtualisation data
  - Always thinks it is on bare metal
  - Makes bad choices because of missing information
- No direct support for containers
  - For example, Docker

4. Java performance, hard to diagnose

You have to combine metrics from Java with metrics from:

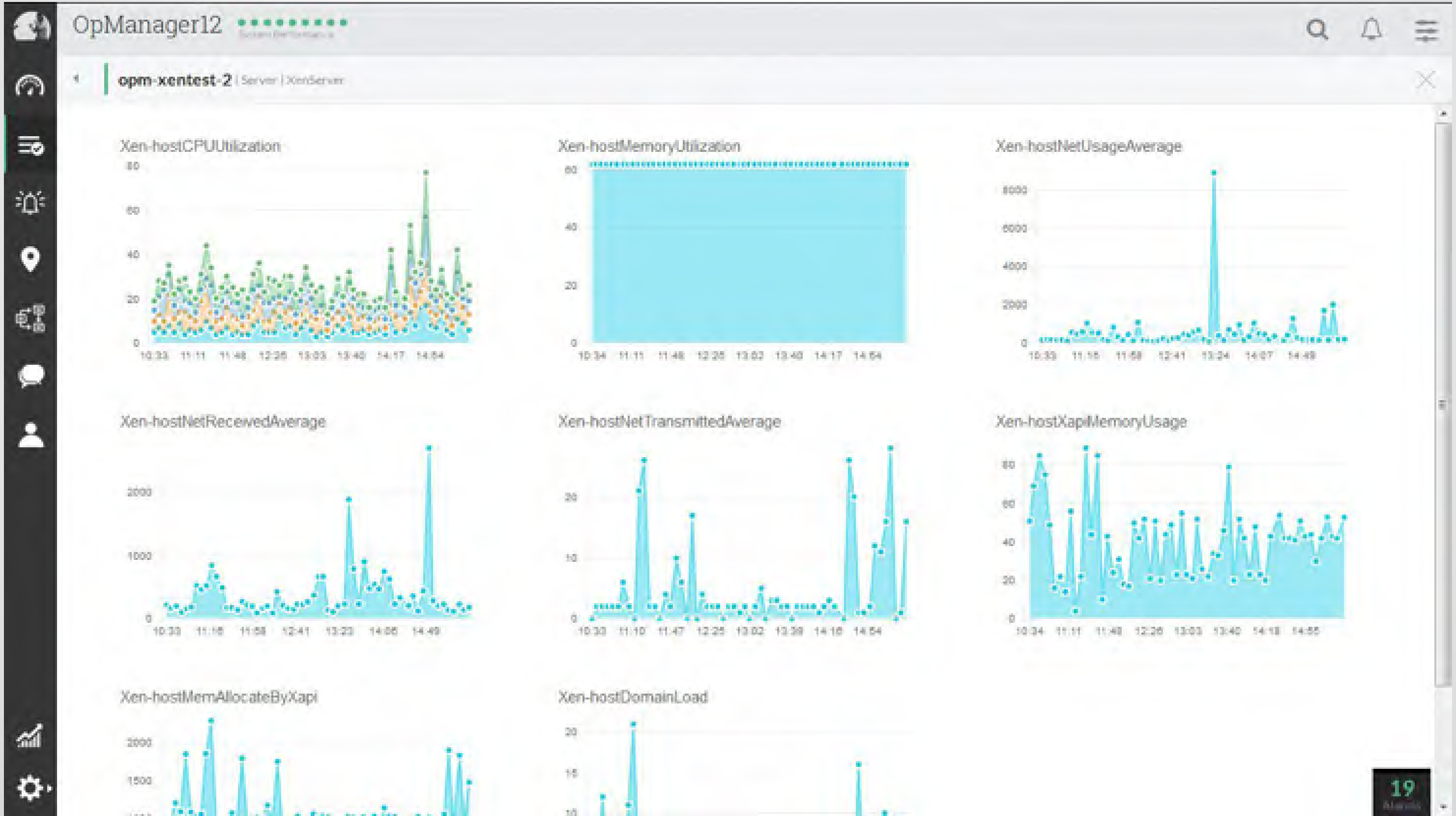
- CPU, Memory
- Disk I/O, Network I/O
- Virtualisation, Containers

Metrics create a Big Data Problem

Because we do not know what we are
looking for, we try to collect billions of points of data

There is a large cost to collecting, transmitting and storing metrics data

Example of a lot of metrics data



Java Diagnosis - GC Example

```
14.896: [GC 14.896: [ParNew
Desired survivor size 1343488 bytes, new threshold 4 (max 4)
- age   1:     181872 bytes,     181872 total
- age   2:     374976 bytes,     556848 total
- age   3:     216304 bytes,     773152 total
- age   4:     129048 bytes,     902200 total
: 16963K->884K(18624K), 0.0017349 secs] 66634K->50555K(81280K), 0.0018305 secs]
```

Pause

Heap Occupancy Before and after

Heap Size

Java Diagnosis - Threads Example

```
"BLOCKED_TEST pool-1-thread-1" prio=6 tid=0x0000000006904800 nid=0x28f4 runnable [0x00000000078
   java.lang.Thread.State: RUNNABLE
                at java.io.FileOutputStream.writeBytes(Native Method)
                at java.io.FileOutputStream.write(FileOutputStream.java:282)
                at java.io.BufferedOutputStream.flushBuffer(BufferedOutputStream.java:65)
                at java.io.BufferedOutputStream.flush(BufferedOutputStream.java:123)
                - locked <0x0000000780a31778> (a java.io.BufferedOutputStream)
                at java.io.PrintStream.write(PrintStream.java:432)
                - locked <0x0000000780a04118> (a java.io.PrintStream)
                at sun.nio.cs.StreamEncoder.writeBytes(StreamEncoder.java:202)
                at sun.nio.cs.StreamEncoder.implFlushBuffer(StreamEncoder.java:272)
                at sun.nio.cs.StreamEncoder.flushBuffer(StreamEncoder.java:85)
                - locked <0x0000000780a040c0> (a java.io.OutputStreamWriter)
                at java.io.OutputStreamWriter.flushBuffer(OutputStreamWriter.java:168)
                at java.io.PrintStream.newLine(PrintStream.java:496)
                - locked <0x0000000780a04118> (a java.io.PrintStream)
                at java.io.PrintStream.println(PrintStream.java:687)
                - locked <0x0000000780a04118> (a java.io.PrintStream)
                at com.nbp.theplatform.threaddump.ThreadBlockedState.monitorLock(ThreadBlockedS
                - locked <0x0000000780a000b0> (a com.nbp.theplatform.threaddump.ThreadBlockedSt
                at com.nbp.theplatform.threaddump.ThreadBlockedState$1.run(ThreadBlockedState.j
                at java.util.concurrent.ThreadPoolExecutor$Worker.runTask(ThreadPoolExecutor.ja
                at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:9
                at java.lang.Thread.run(Thread.java:662)
```

Java performance is hard to diagnose

The previous 3 slides showed examples of metrics being shown in graphs or in a log file.

This is not as helpful as it could be!

Analytics > Metrics

Humans are now finding it very hard to do proper analysis. We have to:

1. Understand the Laws
2. Understand Hardware, O/S, Java & Code
3. Process billions of points of data

麻雀虽小,五脏俱全

Small as it is, the sparrow has all the vital organs

The Future - Analytics

We think that the future is applying advanced statistics & Machine Learning over metrics

The Future - Analytics > Metrics

I will now show an example of how we take metrics about Garbage Collection and reduce them to come up with some analysis.

jClarity censum

Tenured Occupancy After GC

PAUSE TIME

- High Pause Times
- Application Throughput

SYSTEM GC

- Calls to System.gc()

CPU USAGE

- High kernel times

LOG INFORMATION

- Log duration 13 hours 36 minutes

GRAPHS AND DATA

Summary

HEAP USAGE

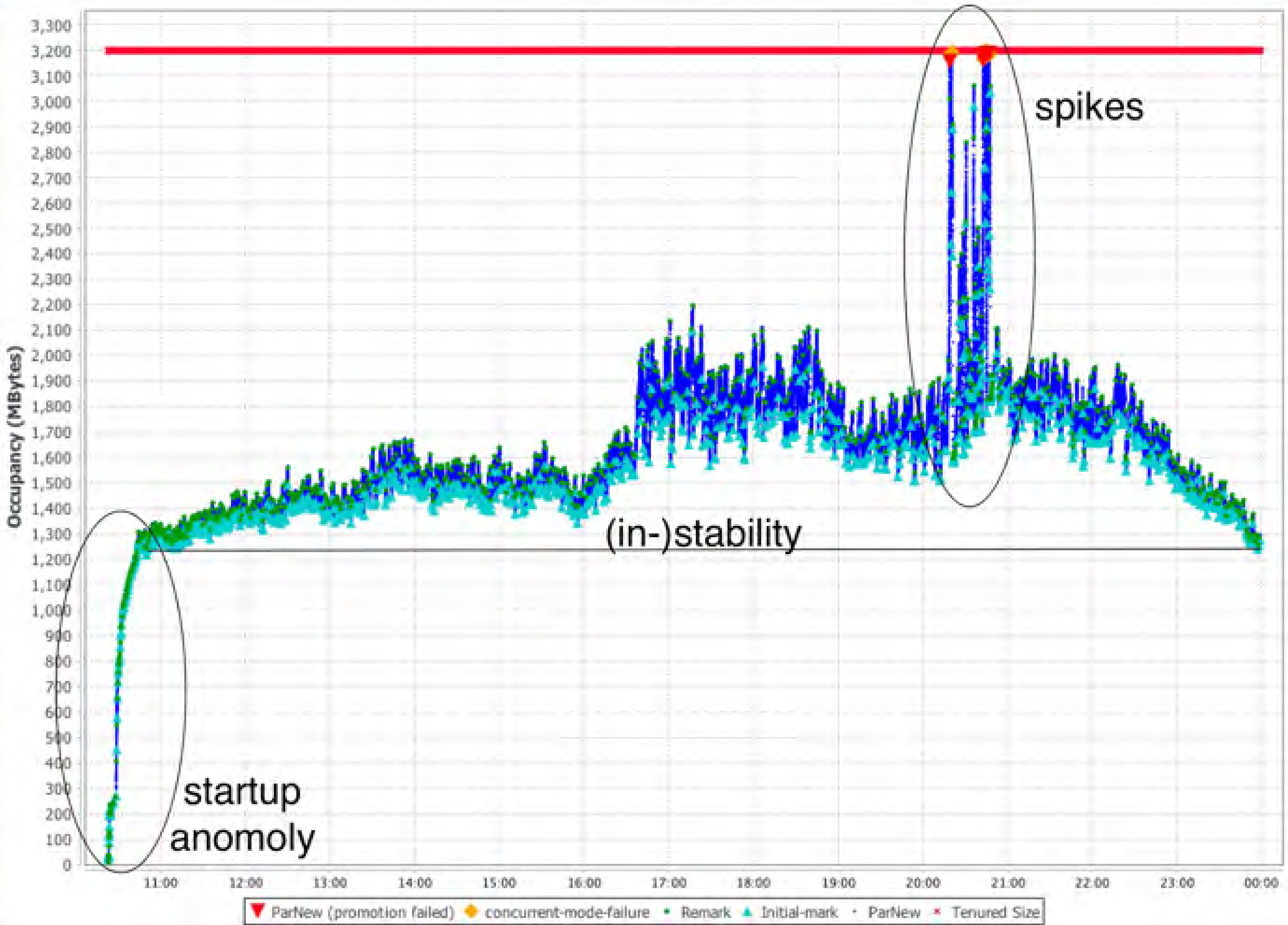
- Heap After GC
- Heap Before GC
- Tenured After GC
- Tenured Before GC
- Aggregate Allocations
- Resident Set Size
- Resident Set Size Experimental
- Resident Set Size Scratch Pad

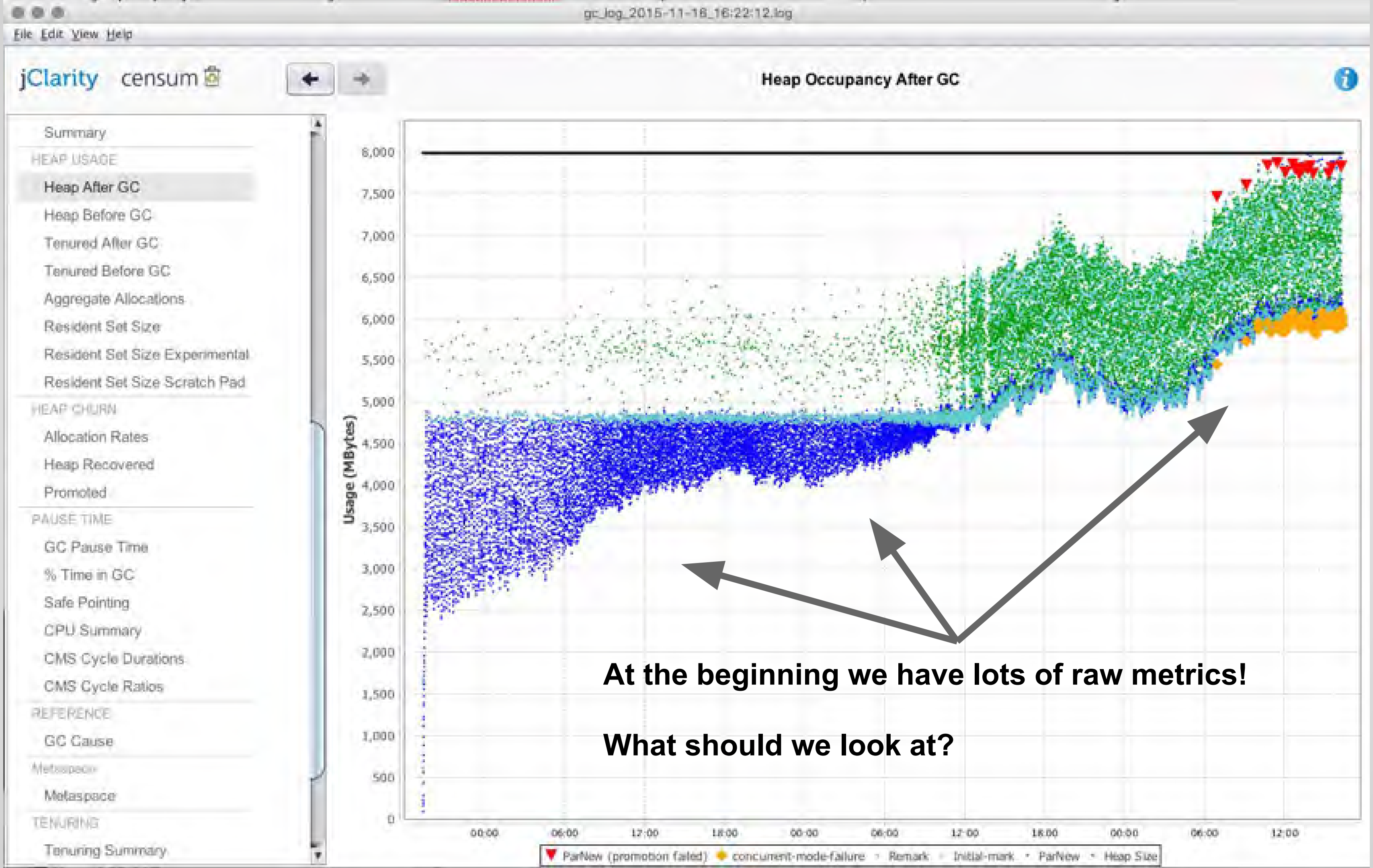
HEAP CHURN

- Allocation Rates
- Heap Recovered
- Promoted

PAUSE TIME

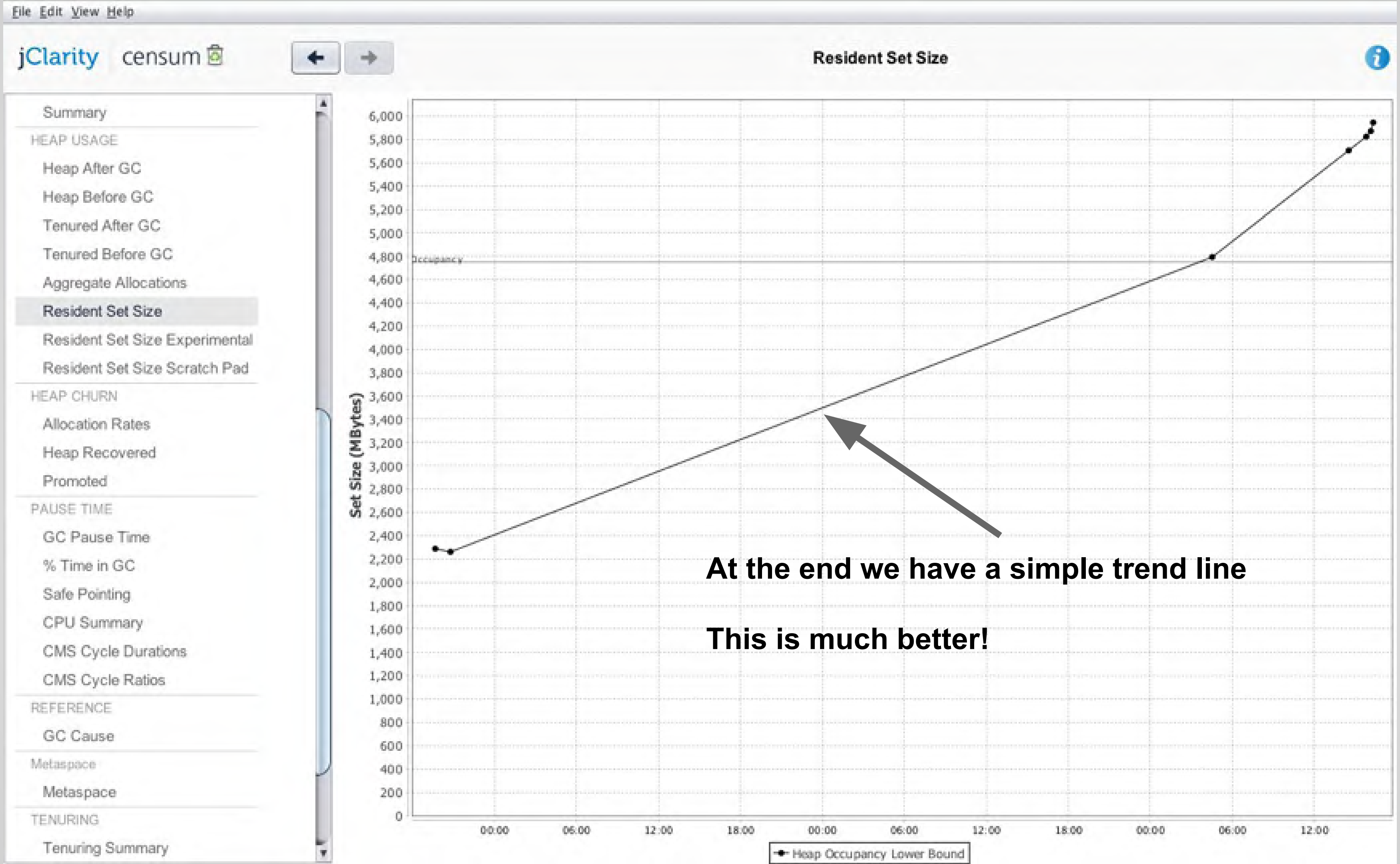
- GC Pause Time



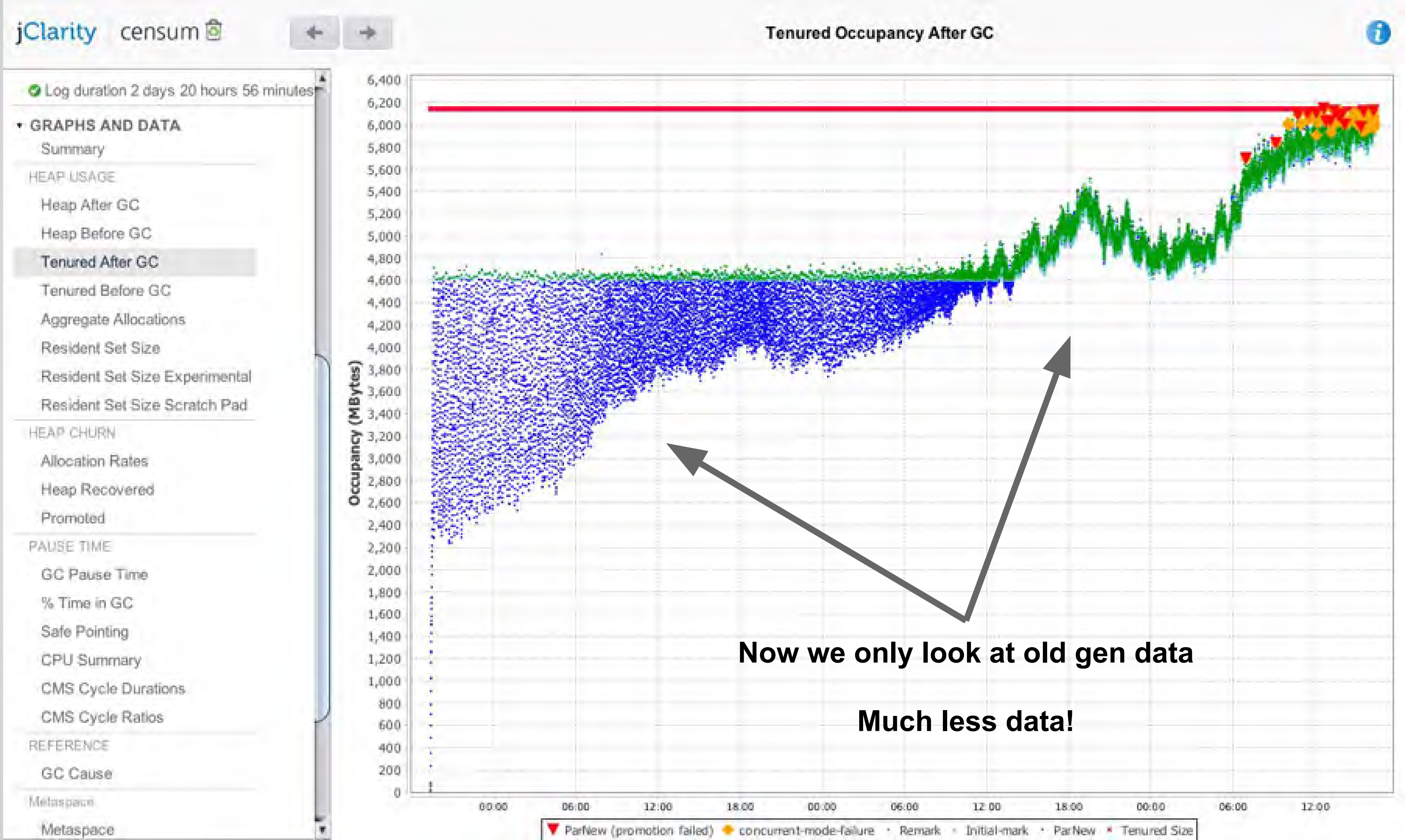


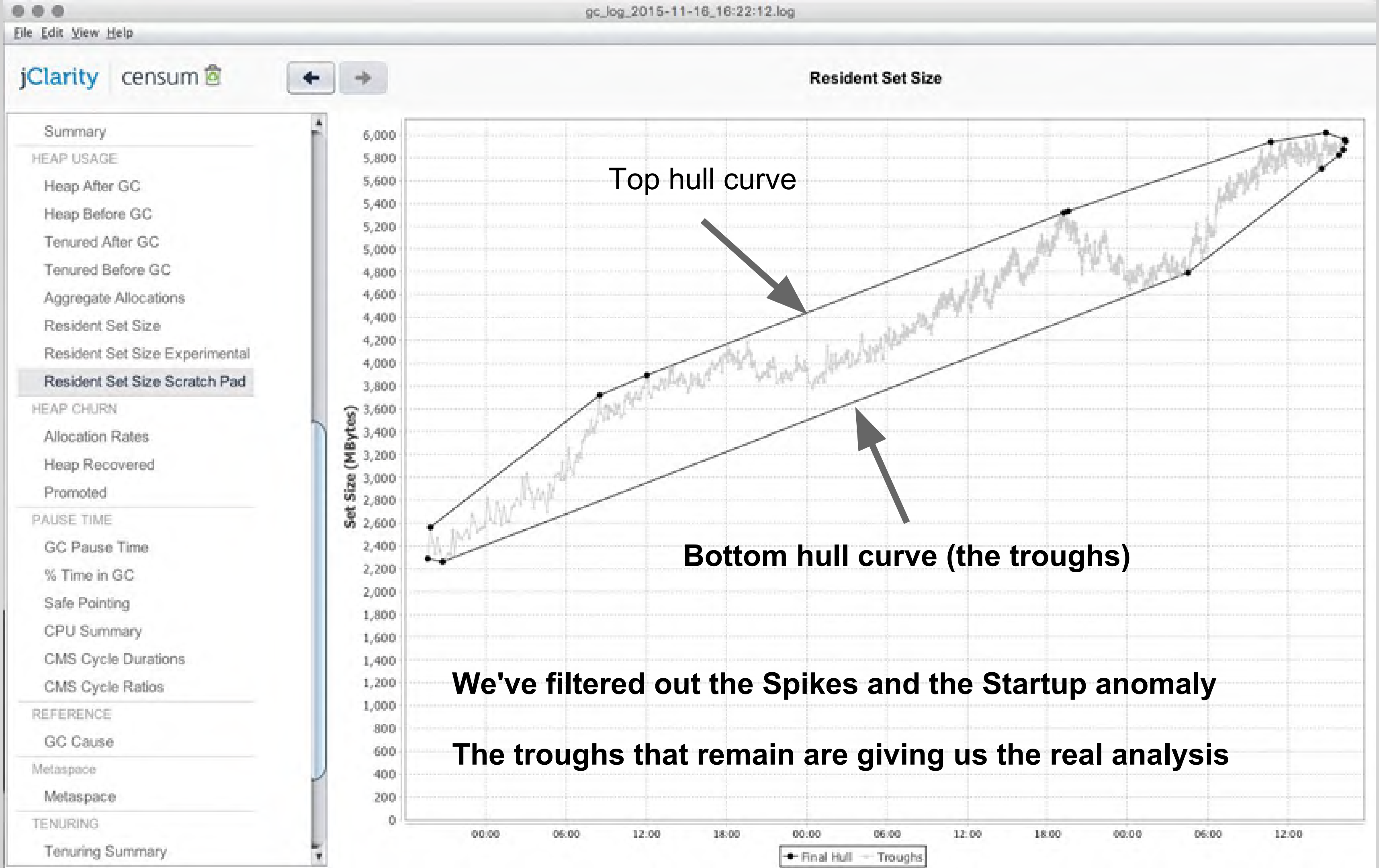
At the beginning we have lots of raw metrics!

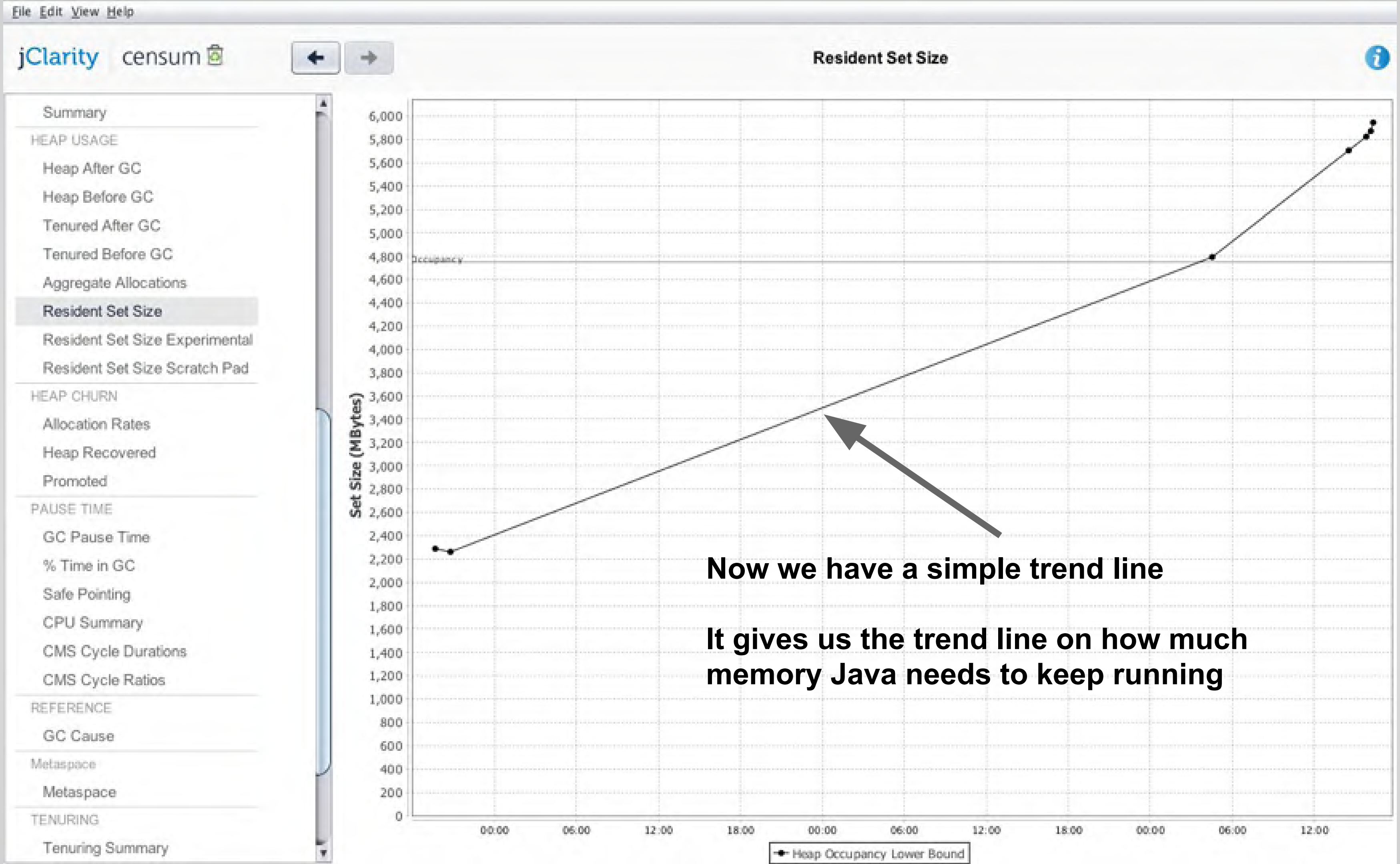
What should we look at?


File Edit View Help
jClarity censum
Resident Set Size
Summary
HEAP USAGE
Heap After GC
Heap Before GC
Tenured After GC
Tenured Before GC
Aggregate Allocations
Resident Set Size
Resident Set Size Experimental
Resident Set Size Scratch Pad
HEAP CHURN
Allocation Rates
Heap Recovered
Promoted
PAUSE TIME
GC Pause Time
% Time in GC
Safe Pointing
CPU Summary
CMS Cycle Durations
CMS Cycle Ratios
REFERENCE
GC Cause
Metaspace
Metaspace
TENURING
Tenuring Summary
Set Size (MBytes)
6,000
5,800
5,600
5,400
5,200
5,000
4,800
4,600
4,400
4,200
4,000
3,800
3,600
3,400
3,200
3,000
2,800
2,600
2,400
2,200
2,000
1,800
1,600
1,400
1,200
1,000
800
600
400
200
0
Occupancy
00:00
06:00
12:00
18:00
00:00
06:00
12:00
18:00
00:00
06:00
12:00
Heap Occupancy Lower Bound
At the end we have a simple trend line
This is much better!


gc_log_2015-11-18_16:22:12.log
File Edit View Help
jClarity censum
Heap Occupancy After GC
Summary
HEAP USAGE
Heap After GC
Heap Before GC
Tenured After GC
Tenured Before GC
Aggregate Allocations
Resident Set Size
Resident Set Size Experimental
Resident Set Size Scratch Pad
HEAP CHURN
Allocation Rates
Heap Recovered
Promoted
PAUSE TIME
GC Pause Time
% Time in GC
Safe Pointing
CPU Summary
CMS Cycle Durations
CMS Cycle Ratios
REFERENCE
GC Cause
Metaspace
TENURING
Tenuring Summary
Usage (MBytes)
8,000
7,500
7,000
6,500
6,000
5,500
5,000
4,500
4,000
3,500
3,000
2,500
2,000
1,500
1,000
500
0
00:00
06:00
12:00
18:00
00:00
06:00
12:00
18:00
00:00
06:00
12:00
ParNew (promotion failed)
concurrent-mode-failure
Remark
Initial-mark
ParNew
Heap Size
What should we look at?
There is both young gen and old gen data

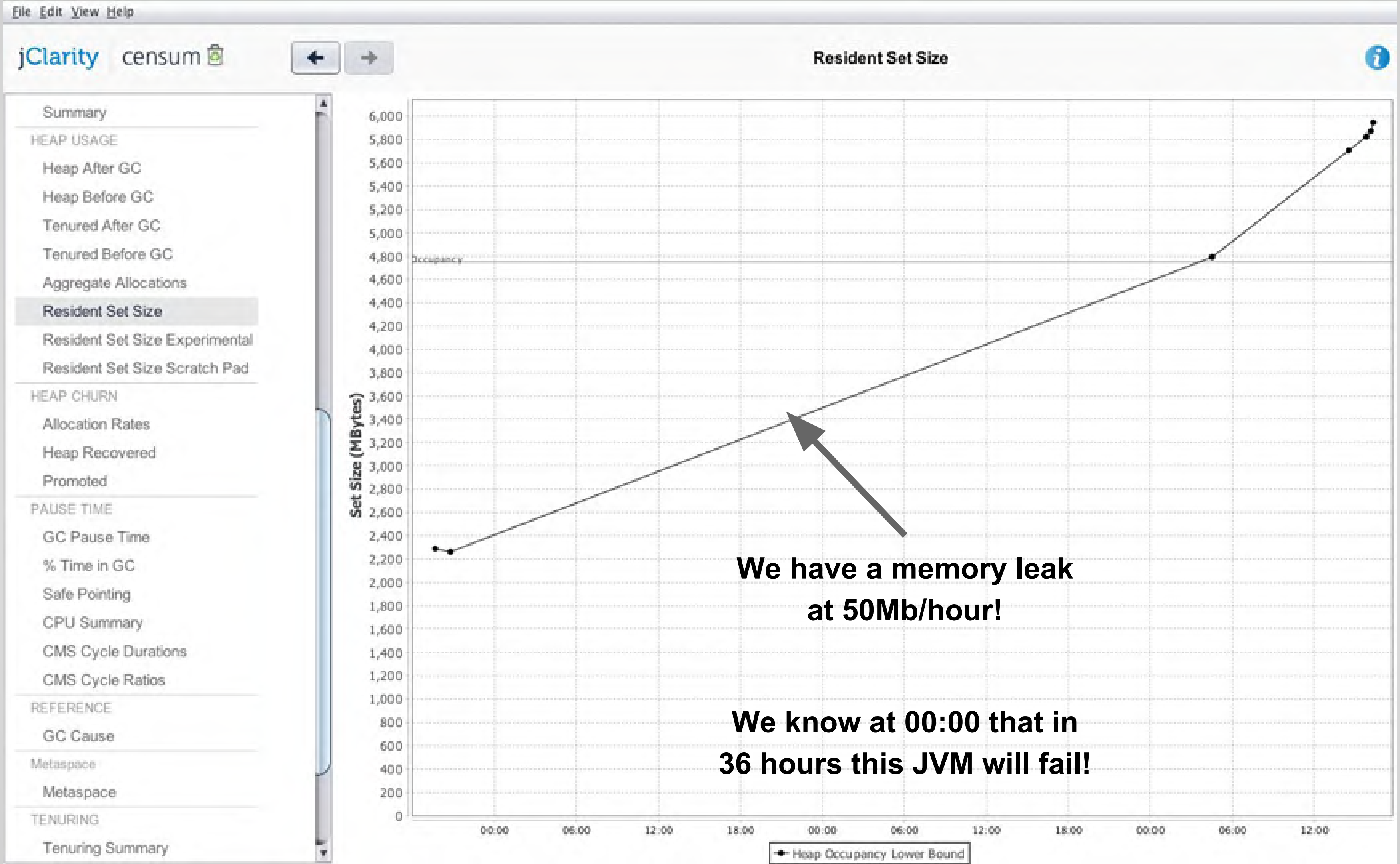

jClarity censum
Tenured Occupancy After GC
Log duration 2 days 20 hours 56 minutes
GRAPHS AND DATA
Summary
HEAP USAGE
Heap After GC
Heap Before GC
Tenured After GC
Tenured Before GC
Aggregate Allocations
Resident Set Size
Resident Set Size Experimental
Resident Set Size Scratch Pad
HEAP CHURN
Allocation Rates
Heap Recovered
Promoted
PAUSE TIME
GC Pause Time
% Time in GC
Safe Pointing
CPU Summary
CMS Cycle Durations
CMS Cycle Ratios
REFERENCE
GC Cause
Metaspace
Metaspace
Occupancy (MBytes)
6,400
6,200
6,000
5,800
5,600
5,400
5,200
5,000
4,800
4,600
4,400
4,200
4,000
3,800
3,600
3,400
3,200
3,000
2,800
2,600
2,400
2,200
2,000
1,800
1,600
1,400
1,200
1,000
800
600
400
200
0
00:00
06:00
12:00
18:00
00:00
06:00
12:00
18:00
00:00
06:00
12:00
ParNew (promotion failed)
concurrent-mode-failure
Remark
Initial-mark
ParNew
Tenured Size
Now we only look at old gen data
Much less data!


gc_log_2015-11-16_16:22:12.log
File Edit View Help
jClarity censum
Resident Set Size
Summary
HEAP USAGE
Heap After GC
Heap Before GC
Tenured After GC
Tenured Before GC
Aggregate Allocations
Resident Set Size
Resident Set Size Experimental
Resident Set Size Scratch Pad
HEAP CHURN
Allocation Rates
Heap Recovered
Promoted
PAUSE TIME
GC Pause Time
% Time in GC
Safe Pointing
CPU Summary
CMS Cycle Durations
CMS Cycle Ratios
REFERENCE
GC Cause
Metaspace
Metaspace
TENURING
Tenuring Summary
Set Size (MBytes)
Top hull curve
Bottom hull curve (the troughs)
We've filtered out the Spikes and the Startup anomaly
The troughs that remain are giving us the real analysis
Final Hull
Troughs


File Edit View Help
jClarity censum
Resident Set Size
Summary
HEAP USAGE
Heap After GC
Heap Before GC
Tenured After GC
Tenured Before GC
Aggregate Allocations
Resident Set Size
Resident Set Size Experimental
Resident Set Size Scratch Pad
HEAP CHURN
Allocation Rates
Heap Recovered
Promoted
PAUSE TIME
GC Pause Time
% Time in GC
Safe Pointing
CPU Summary
CMS Cycle Durations
CMS Cycle Ratios
REFERENCE
GC Cause
Metaspace
Metaspace
TENURING
Tenuring Summary
Set Size (MBytes)
6,000
5,800
5,600
5,400
5,200
5,000
4,800
4,600
4,400
4,200
4,000
3,800
3,600
3,400
3,200
3,000
2,800
2,600
2,400
2,200
2,000
1,800
1,600
1,400
1,200
1,000
800
600
400
200
0
Occupancy
00:00
06:00
12:00
18:00
00:00
06:00
12:00
18:00
00:00
06:00
12:00
Heap Occupancy Lower Bound
Now we have a simple trend line
It gives us the trend line on how much memory Java needs to keep running

Now we can perform analytics!

- We removed most of the data!
  - We no longer have a Big Data problem
- We have made the information much simpler
- We can now perform analytics!


File Edit View Help
jClarity censum
Resident Set Size
Summary
HEAP USAGE
Heap After GC
Heap Before GC
Tenured After GC
Tenured Before GC
Aggregate Allocations
Resident Set Size
Resident Set Size Experimental
Resident Set Size Scratch Pad
HEAP CHURN
Allocation Rates
Heap Recovered
Promoted
PAUSE TIME
GC Pause Time
% Time in GC
Safe Pointing
CPU Summary
CMS Cycle Durations
CMS Cycle Ratios
REFERENCE
GC Cause
Metaspace
Metaspace
TENURING
Tenuring Summary
Set Size (MBytes)
6,000
5,800
5,600
5,400
5,200
5,000
4,800
4,600
4,400
4,200
4,000
3,800
3,600
3,400
3,200
3,000
2,800
2,600
2,400
2,200
2,000
1,800
1,600
1,400
1,200
1,000
800
600
400
200
0
Occupancy
00:00
06:00
12:00
18:00
00:00
06:00
12:00
18:00
00:00
06:00
12:00
Heap Occupancy Lower Bound
We have a memory leak
at 50Mb/hour!
We know at 00:00 that in
36 hours this JVM will fail!

Now we can perform analytics!

- We can tell you it is a memory leak
  - 50Mb / hour!

- We can **predictively** tell you when your JVM will have an OOME

- There are many other analyses possible...

Conclusion

- Hardware has changed
- Remember your performance laws!
- Java is not optimised for the new world
- It is hard to diagnose Java with only metrics
- The future is Analytics!

Credits

Kirk Pepperdine - jClarity (CTO)
John Oliver - jClarity (Chief Scientist)
Ben Evans - jClarity (Tech Fellow)
Kerry Kenneally - jClarity (UI/Ux)

谢谢

jClarity

www.jclarity.com - martijn@jclarity.com