The Missing Benchmark Metric: Memory

#java #benchmark #JMH #memory #metrics #GC
#caching #optimizethewhole #bigpicture

2nd February 2019
Java Dev Room, FOSDEM, Brussels

Jens Wilke
@cruftex
cruftex.net

About Me

- Jens Wilke
- Performance Fan(atic)
- Author of cache2k
- 70+ answered questions on StackOverflow about Caching
- JCache / JSR107 Contributor
- Tiny contributions to JMH
- @cruftex / cruftex.net

cache2k

https://cache2k.org

Content

- Why?
- How to gather memory metrics?
- How (not) to gather memory metrics from a running JMH benchmark?
- Real benchmark results …
  …. to get a feeling whether this is worthwhile
- Happyness?!

- Please ask questions right away!

Motivation

- Compare different caching libraries in identical scenario (access sequence, cache size (entries!), CPU and JDK) with JMH
- Primary result: Throughput in **ops/s**
- But: Compare **caching libraries** and ignore memory consumption?!
- Space vs. Time trade off
- But:
  no mechanism in JMH to record memory usage metrics (yet…)

Metric: Object Graph Traversing

- EHCache sizeof
- Java Agent for Memory Measurements

Pro:

- No actual usage long running benchmark needed, just fill up the data scructures
- No complex setup (dedicated hardware)

Con:

- How to choose the root object / traverse the whole heap?
- Static result
- heap objects only
- Needs time

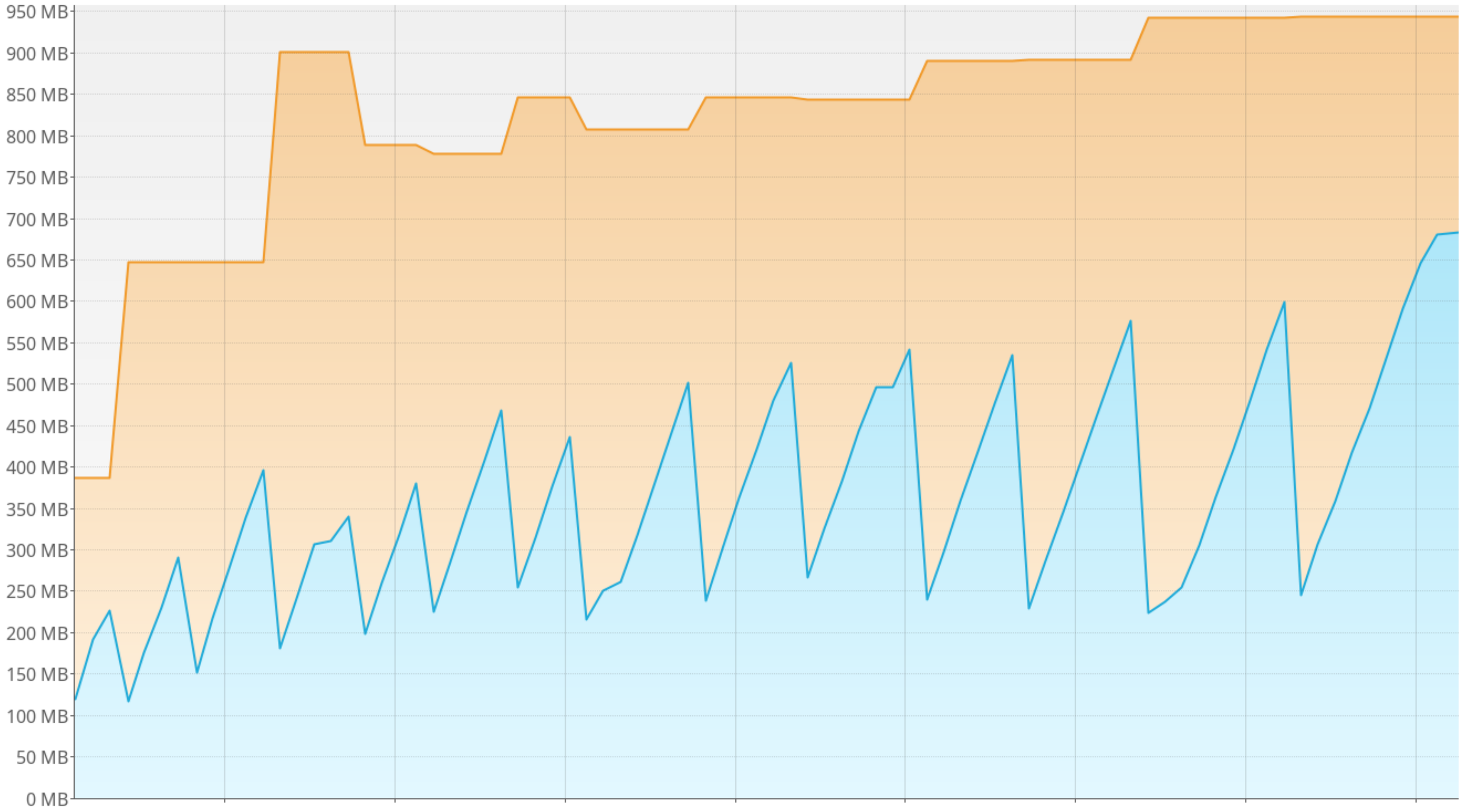
Metric: Heap Dump / Heap Historgram

- e.g. use jmap to get a heap dump or histogram

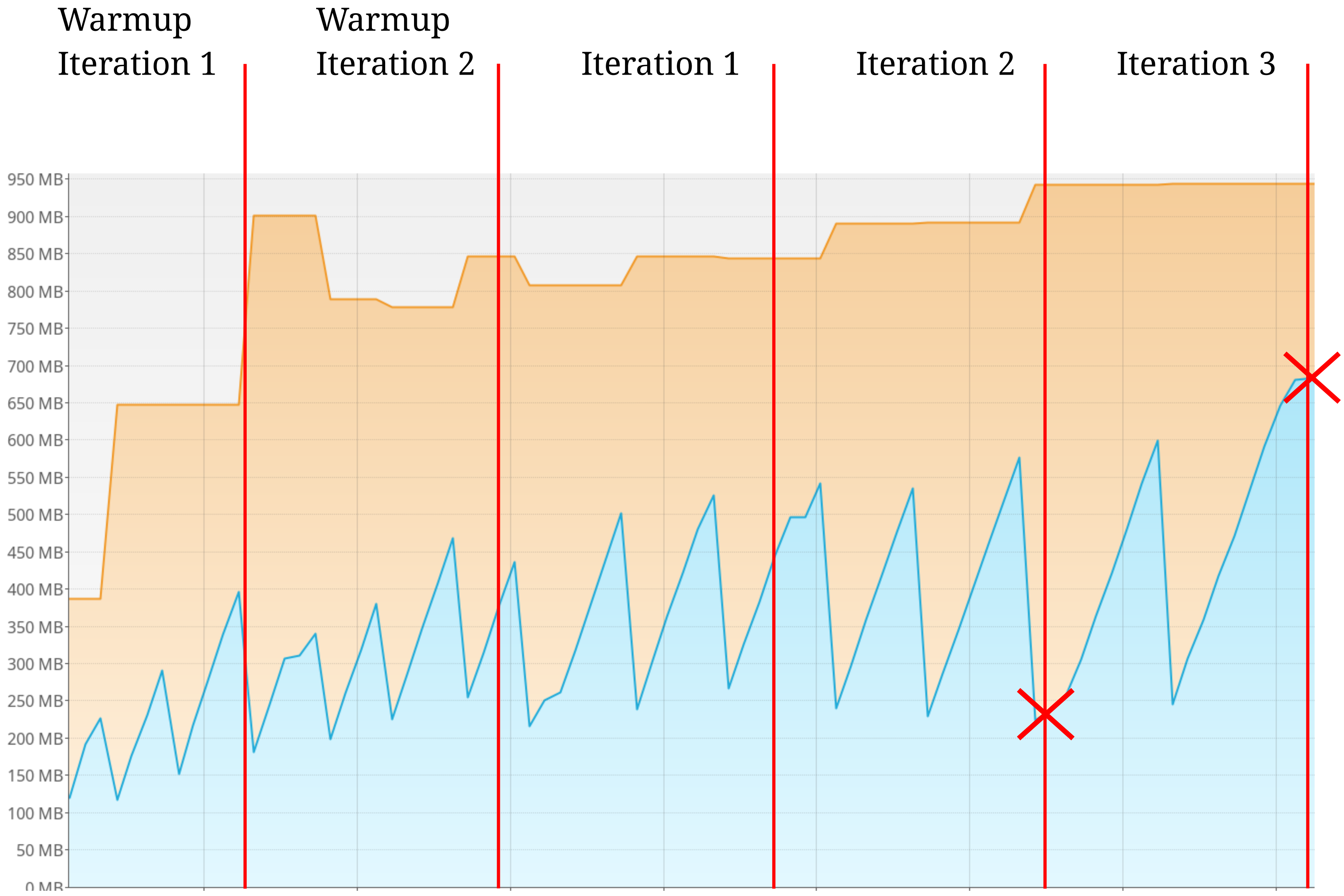
  Pro:
  - Same as before, we don‘t need a real benchmark run
  - Accurate value of used heap space
  - Additional information in the histogram or dump

  Con:
  - Costly
  - Static value of heap objects only

Let it Run!


950 MB
900 MB
850 MB
800 MB
750 MB
700 MB
650 MB
600 MB
550 MB
500 MB
450 MB
400 MB
350 MB
300 MB
250 MB
200 MB
150 MB
100 MB
50 MB
0 MB

How to Extract a (single) Metric?



Warmup Iteration 1

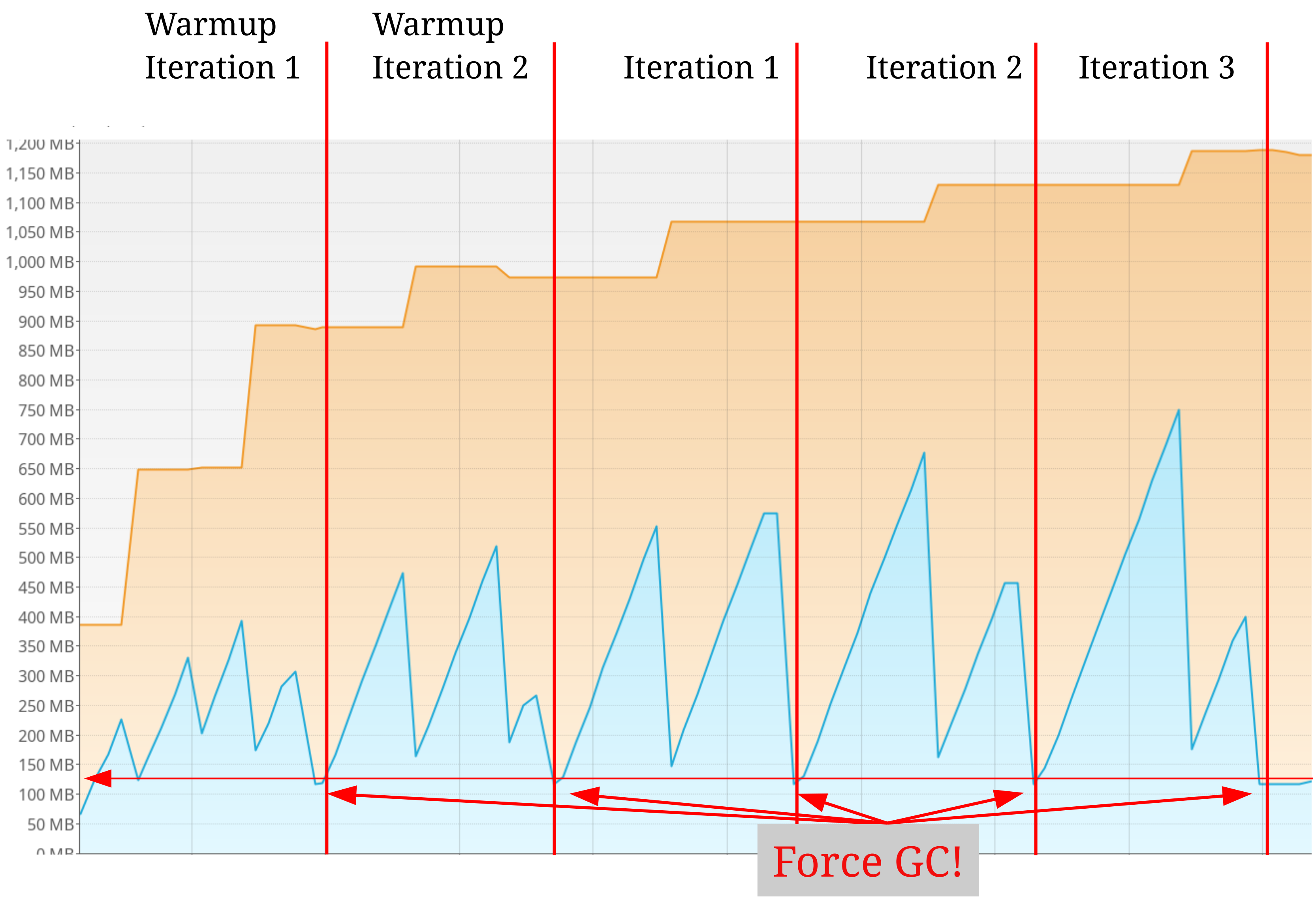
Warmup Iteration 2

Iteration 1

Iteration 2

Iteration 3

Fore GC?



Warmup Iteration 1

Warmup Iteration 2

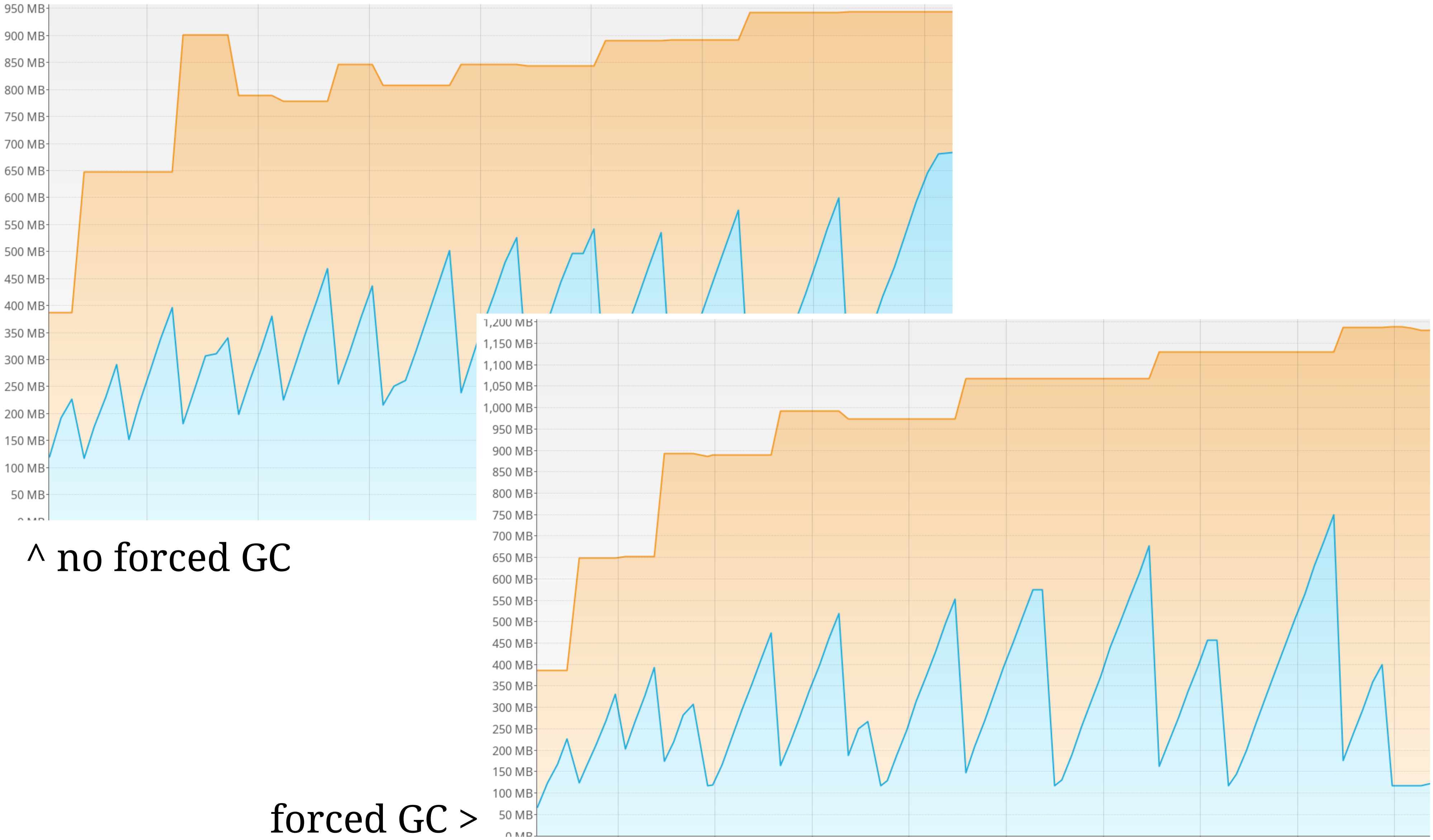
Iteration 1

Iteration 2

Iteration 3

Force GC!

Side to Side Comparison



^ no forced GC

forced GC >

Wait a minute….

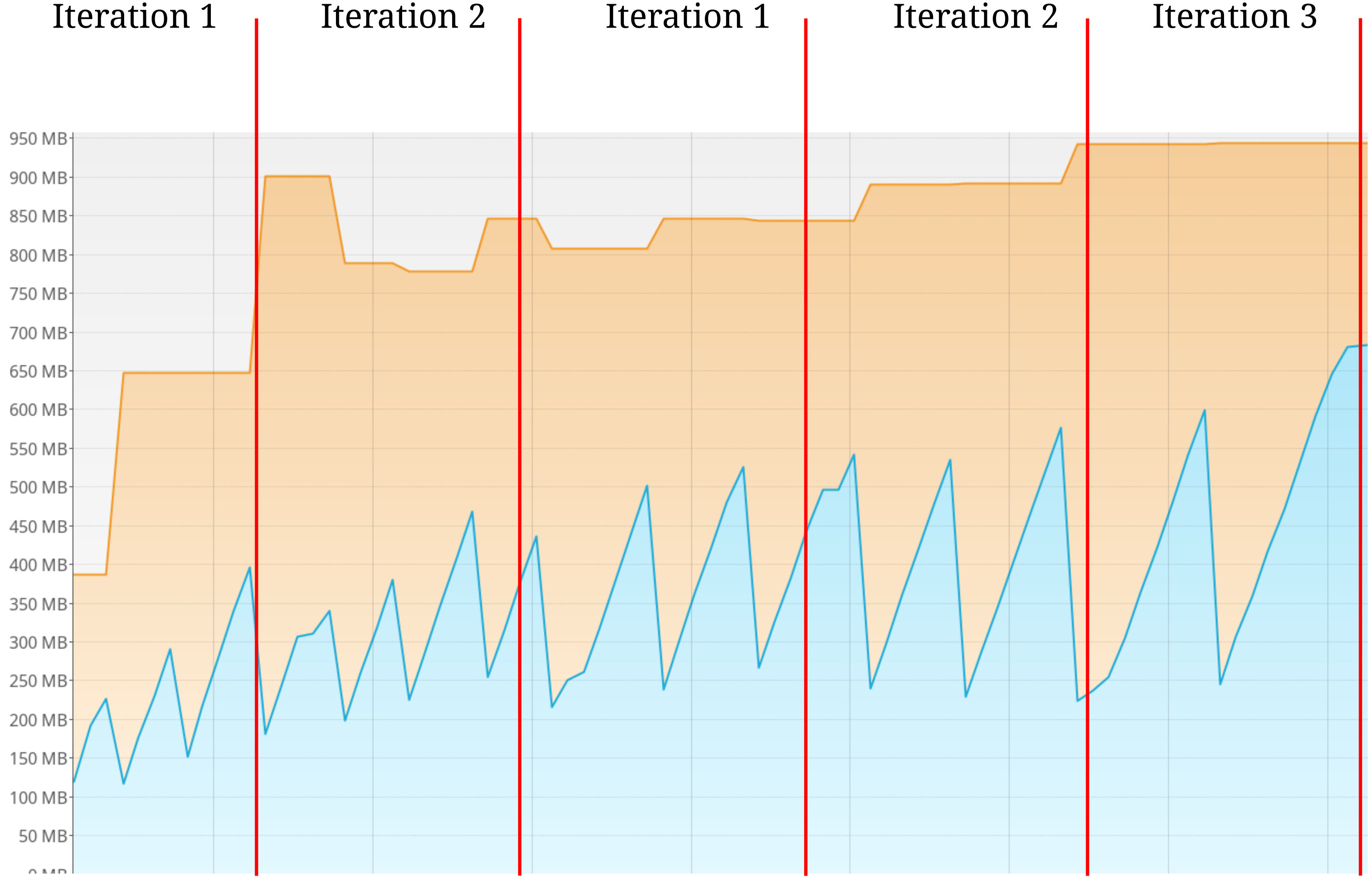
Warmup
Iteration 1

Warmup
Iteration 2

Iteration 1

Iteration 2

Iteration 3



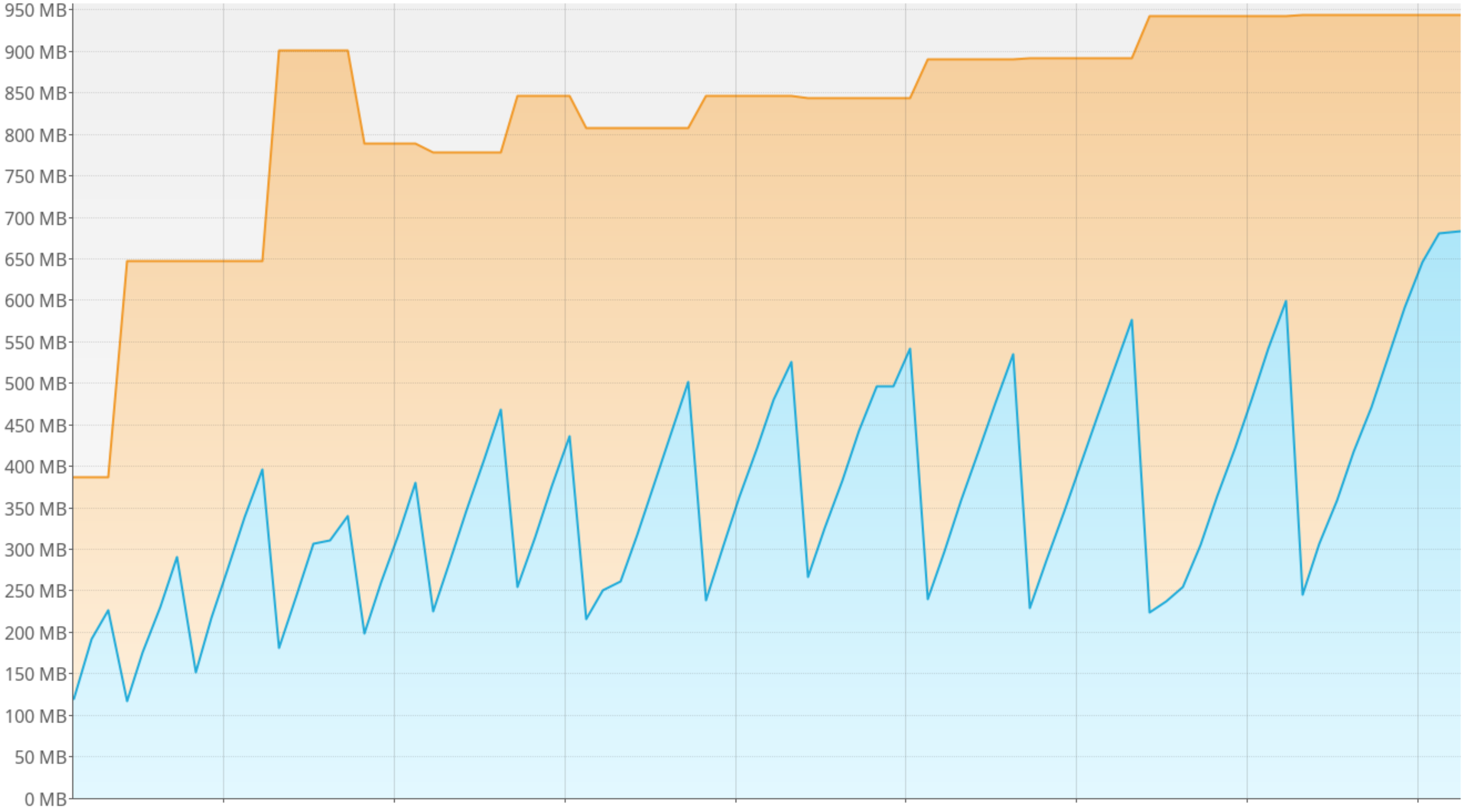
Sidenote:
GC and Micro Benchmarks

- Micro-Microbenchmark: GC might happen during an iteration
  - Single GC occurence causes skewed result
  - Use JMH parameter `-gc true` or Zero GC

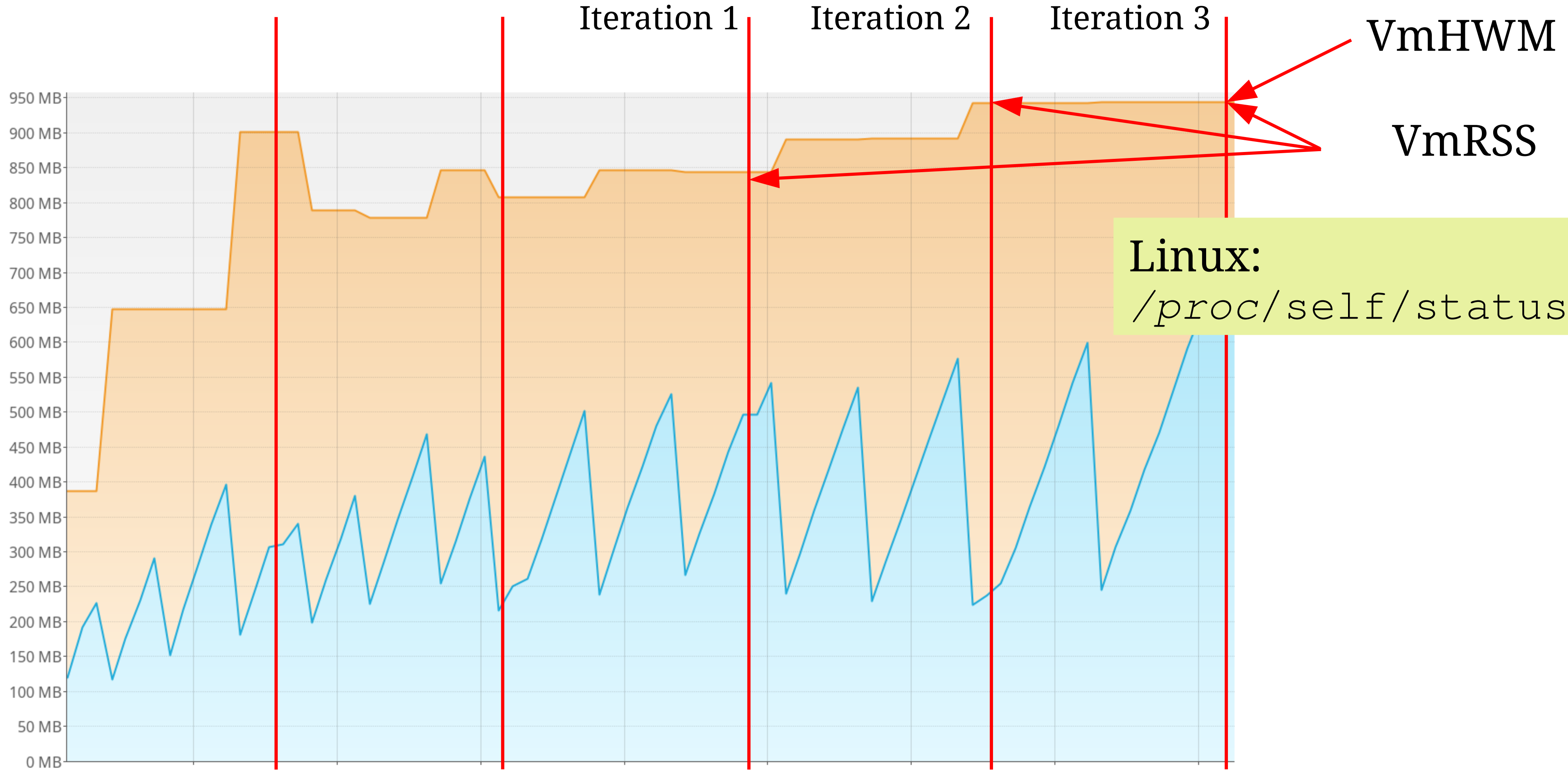
  => Get garbage collector out of the equation
- NotSo-Microbenchmark: GC happens always during an iteration
  - Make sure a lot of GCs happen, JMH parameter: `-prof gc`
  - Increase warmup and iteration time
  - „Know your GC“ ↔ „Get to know your GC“

  => You cannot avoid the GC. Make it go steady.

A Metric? Some Metrics!



Linux OS Metrics: VmRSS, VmHWM



Iteration 1

Iteration 2

Iteration 3

VmHWM

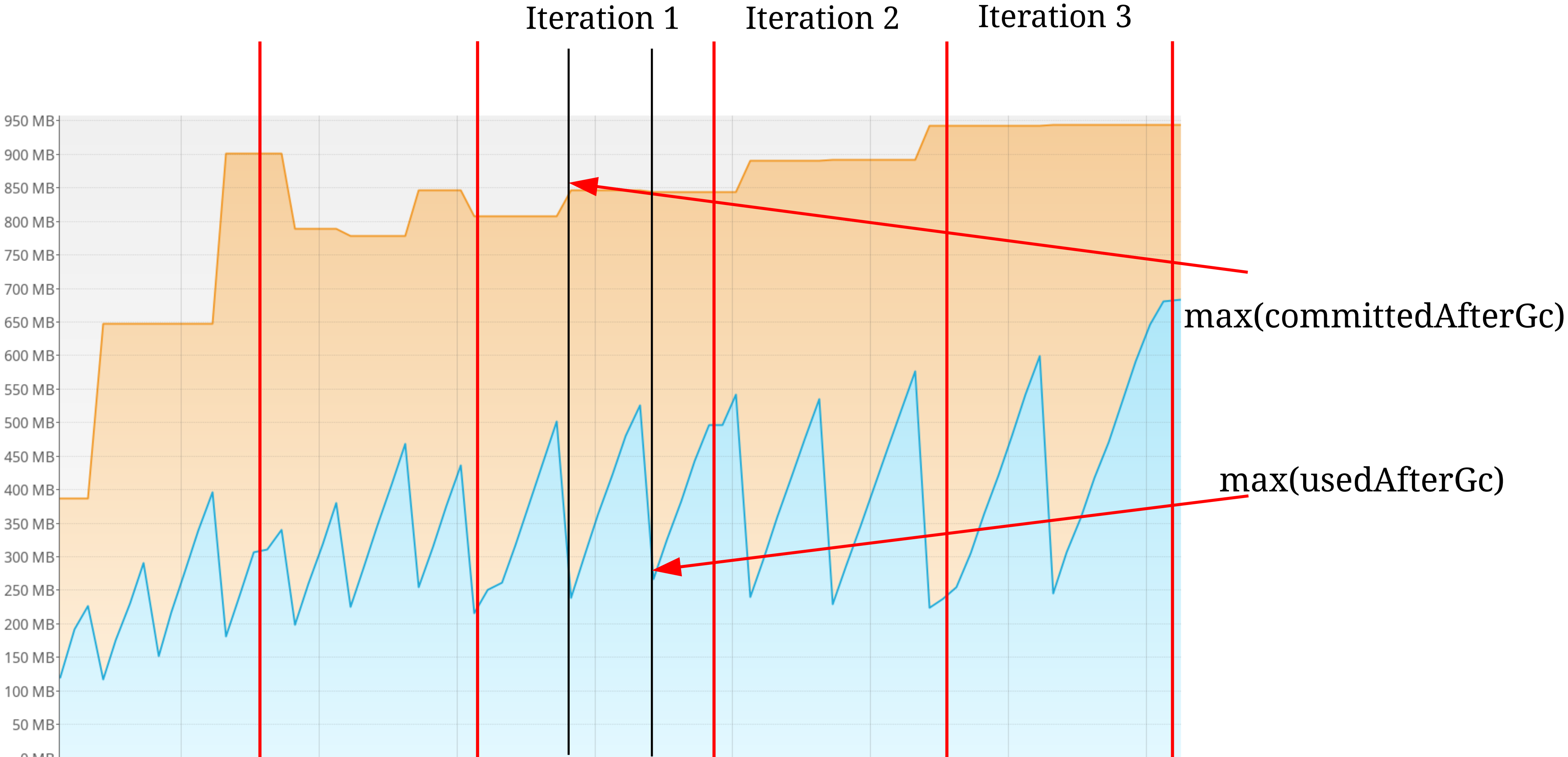
VmRSS

Linux:
`/proc/self/status`

GC Notifications

- Notification after GC run (via `GarbageCollectionNotificationInfo`)
- Contains memory usage before and after GC

  (examples only shows notification in iteration 2)

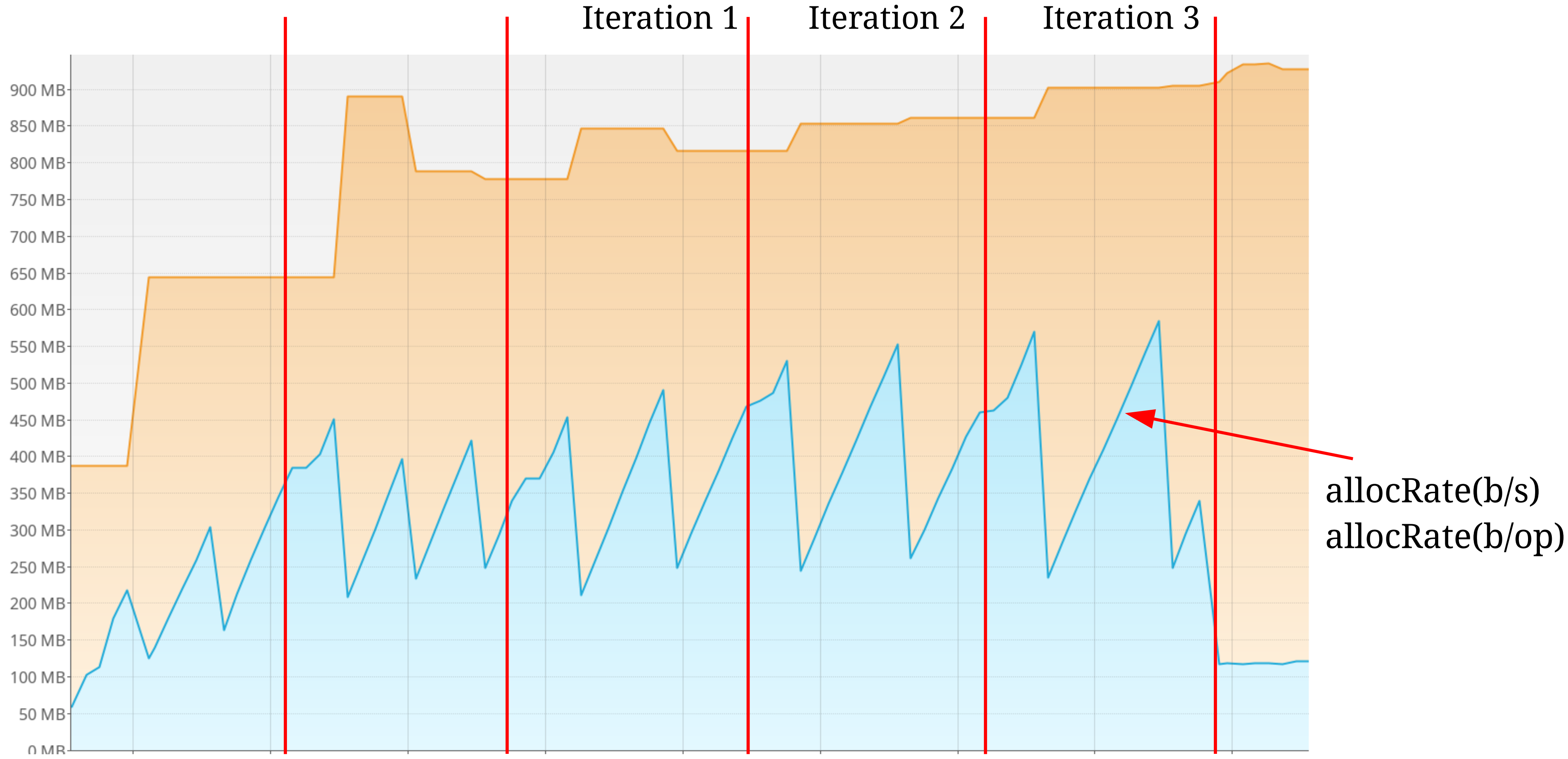


Iteration 1 Iteration 2 Iteration 3

max(committedAfterGc)

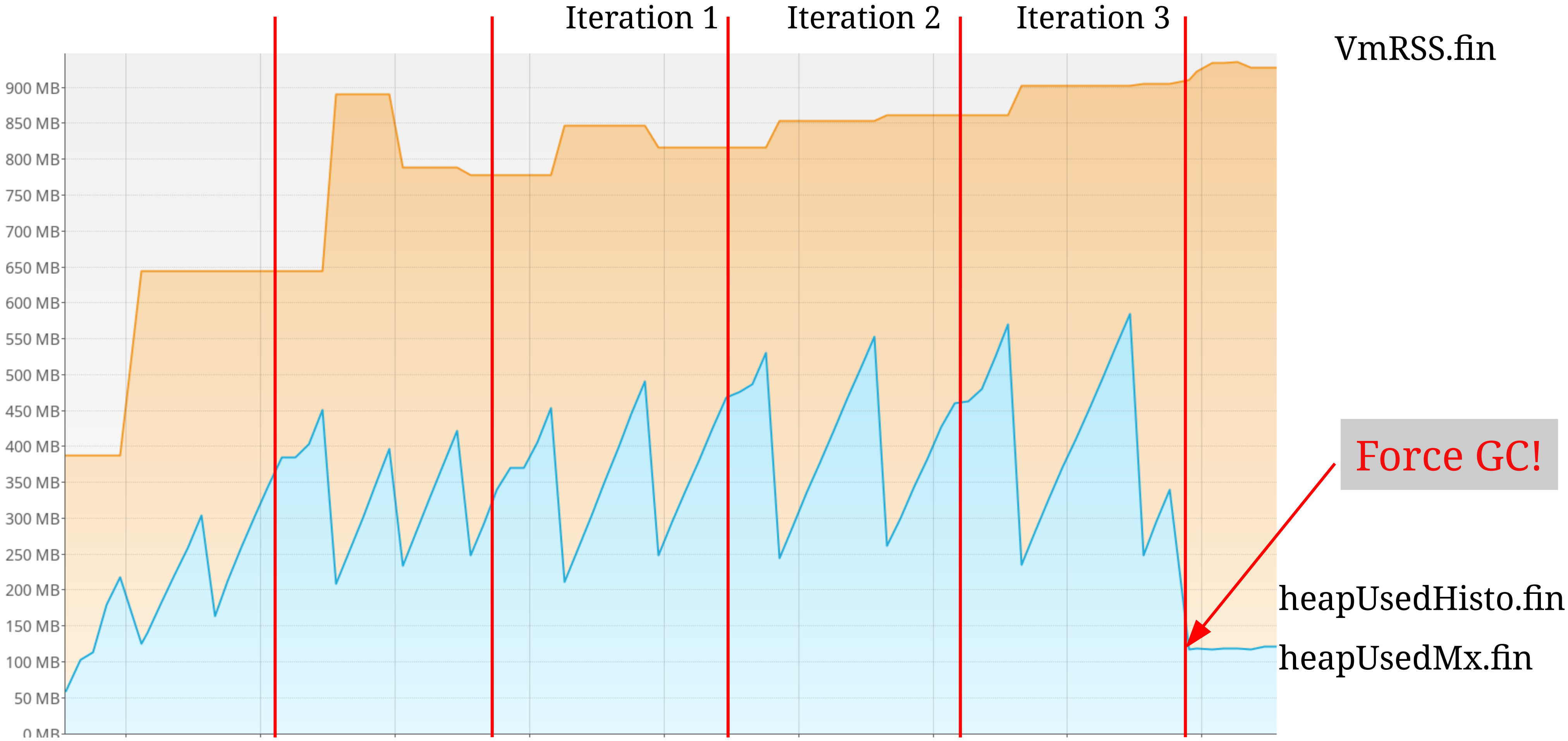
max(usedAfterGc)

Allocation Rate



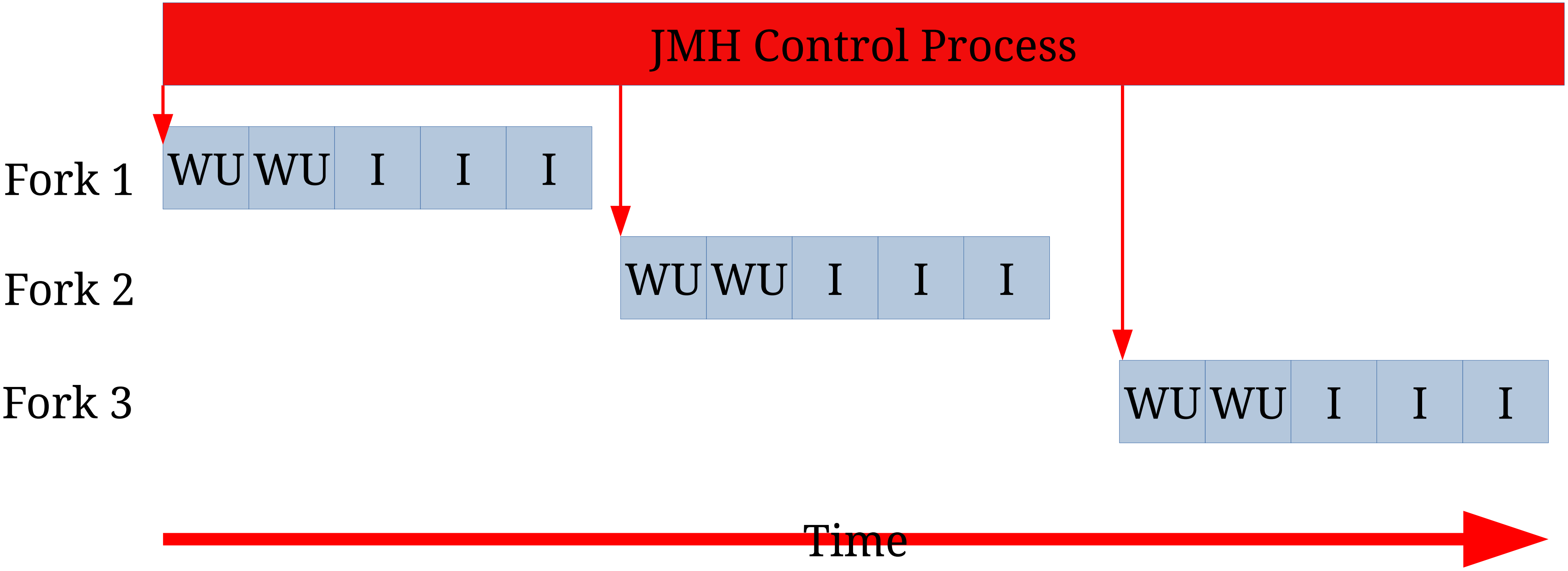
Minimal Memory at the End

- JMH can run multiple VMs after another (forks)
- at end of each fork the invasive collectors could happen (force GC, heap histogram)



Example JMH Running Scheme

- JMH can run multiple VMs after another (forks)
- 9 iterations to collect results
- 3 forks, at end of each fork the invasive collectors happen (force GC, heap histogram)
- Parameters: `-f 3 -wi 3 -i 3`



Results

JMH Benchmark

```
public int factor = 5;
public int entryCount = 100_000;

@State(Scope.Thread)
public static class ThreadState {

  ZipfianPattern pattern;

  @Setup(Level.Iteration)
  public void setup(ZipfianSequenceLoadingBenchmark benchmark) {
    pattern = new ZipfianPattern(benchmark.offsetSeed.nextLong(),
      benchmark.entryCount * benchmark.factor);
  }

}

@Benchmark @BenchmarkMode(Mode.Throughput)
public long operation(ThreadState threadState, HitCountRecorder rec) {
  // TODO: JMH should return the raw number of operations somewhere...
  rec.opCount++;
  Integer v = cache.get(threadState.pattern.next());
  return v;
}

public Integer load(final Integer key) {
  missCount.increment();
  Blackhole.consumeCPU(1000);
  return key * 2 + 11;
}
```

JMH Benchmark

```
public int factor = 5;
public int entryCount = 100_000;

@State(Scope.Thread)
public static class ThreadState {

  ZipfianPattern pattern;

  @Setup(Level.Iteration)
  public void setup(ZipfianSequenceLoadingBenchmark benchmark) {
    pattern = new ZipfianPattern(benchmark.offsetSeed.nextLong(),
      benchmark.entryCount * benchmark.factor);
  }

}

@Benchmark @BenchmarkMode(Mode.Throughput)
public long operation(ThreadState threadState, HitCountRecorder rec) {
  // TODO: JMH should return the raw number of operations somewhere...
  rec.opCount++;
  Integer v = cache.get(threadState.pattern.next());
  return v;
}

public Integer load(final Integer key) {
  missCount.increment();
  Blackhole.consumeCPU(1000);
  return key * 2 + 11;
}
```

(fast) Zipfian sequence generator per thread

Skewed random pattern, yields around 90% hitrate

JMH Benchmark

```
public int factor = 5;
public int entryCount = 100_000;

@State(Scope.Thread)
public static class ThreadState {

  ZipfianPattern pattern;

  @Setup(Level.Iteration)
  public void setup(ZipfianSequenceLoadingBenchmark benchmark) {
    pattern = new ZipfianPattern(benchmark.offsetSeed.nextLong(),
      benchmark.entryCount * benchmark.factor);
  }

}

@Benchmark @BenchmarkMode(Mode.Throughput)
public long operation(ThreadState threadState, HitCountRecorder rec) {
  // TODO: JMH should return the raw number of operations somewhere...
  rec.opCount++;
  Integer v = cache.get(threadState.pattern.next());
  return v;
}

public Integer load(final Integer key) {
  missCount.increment();
  Blackhole.consumeCPU(1000);
  return key * 2 + 11;
}
```

(fast) Zipfian sequence generator per thread

Skewed random pattern, yields around 90% hitrate

Benchmarked operation is cache.get()

On miss, cache will invoke a load function(read-trough)

JMH Benchmark

```
public int factor = 5;
public int entryCount = 100_000;

@State(Scope.Thread)
public static class ThreadState {

  ZipfianPattern pattern;

  @Setup(Level.Iteration)
  public void setup(ZipfianSequenceLoadingBenchmark benchmark) {
    pattern = new ZipfianPattern(benchmark.offsetSeed.nextLong(),
      benchmark.entryCount * benchmark.factor);
  }

}

@Benchmark @BenchmarkMode(Mode.Throughput)
public long operation(ThreadState threadState, HitCountRecorder rec) {
  // TODO: JMH should return the raw number of operations somewhere...
  rec.opCount++;
  Integer v = cache.get(threadState.pattern.next());
  return v;
}

public Integer load(final Inte
  missCount.increment();
  Blackhole.consumeCPU(1000);
  return key * 2 + 11;
}
```

(fast) Zipfian sequence generator per thread

Skewed random pattern, yields around 90% hitrate

Benchmarked operation is cache.get()

On miss, cache will invoke a load function(read-trough)

Load function has penalty, via Blackhole.consumeCPU()

JMH Benchmark

```
public int factor = 5;
public int entryCount = 100_000;

@State(Scope.Thread)
public static class ThreadState {

  ZipfianPattern pattern;

  @Setup(Level.Iteration)
  public void setup(ZipfianSequenceLoadingBenchmark benchmark) {
    pattern = new ZipfianPattern_benchmark.offsetSeed.nextLong(),
      benchmark.entryCount * benchmark.factor);
  }

}

@Benchmark @BenchmarkMode(Mode.Throughput)
public long operation(ThreadState threadState, HitCountRecorder rec) {
  // TODO: JMH should return the raw number of operations somewhere...
  rec.opCount++;
  Integer v = cache.get(threadState.pattern.next());
  return v;
}

public Integer load(final Inte
  missCount.increment();
  Blackhole.consumeCPU(1000);
  return key * 2 + 11;
}
```

(fast) Zipfian sequence generator per thread

Skewed random pattern, yields around 90% hitrate

Benchmarked operation is cache.get()

On miss, cache will invoke a load function(read-trough)

Load function has penalty, via Blackhole.consumeCPU()

=> a lot can happens on cache.get()

=> Cache eviction and autoboxing

Benchmark Setup

Environment:

- CPU: Intel(R) Xeon(R) CPU E3-1240 v5 @ 3.50GHz, **4 physical cores**
  - Benchmarks run with 4 thread, 4 cores (limited via CPU hotplug)
- Oracle JDK 11, PrallelGC and G1
- Ubuntu 18.04

Cache Library Versions:

- Google Guava Cache, Version 26
- Caffeine, Version 2.6.2
- cache2k, Version 1.2.0.Final
- EHCache, Version 3.6.1

Code is at:

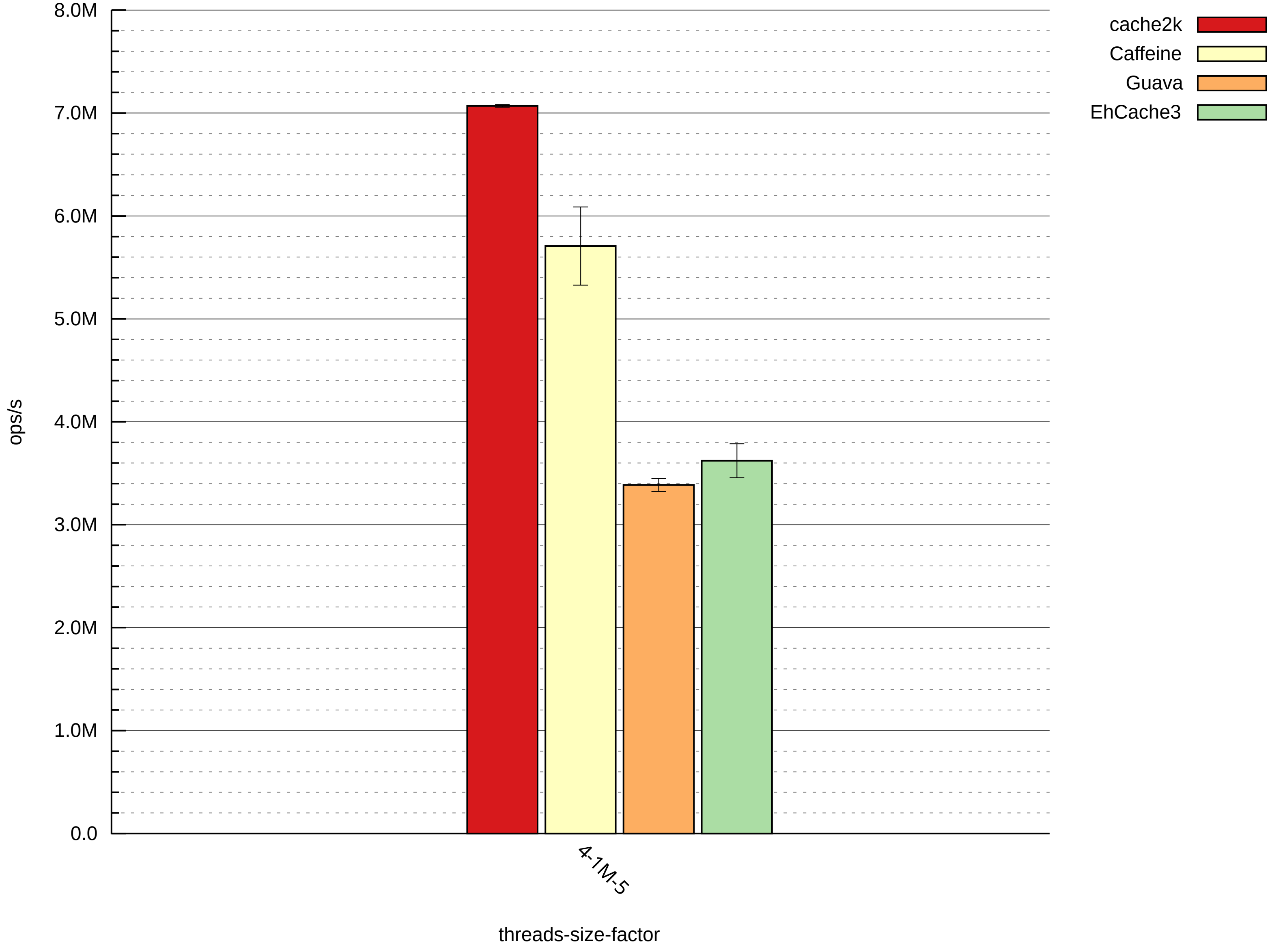
https://github.com/cache2k/cache2k-benchmark

JMH Parameters

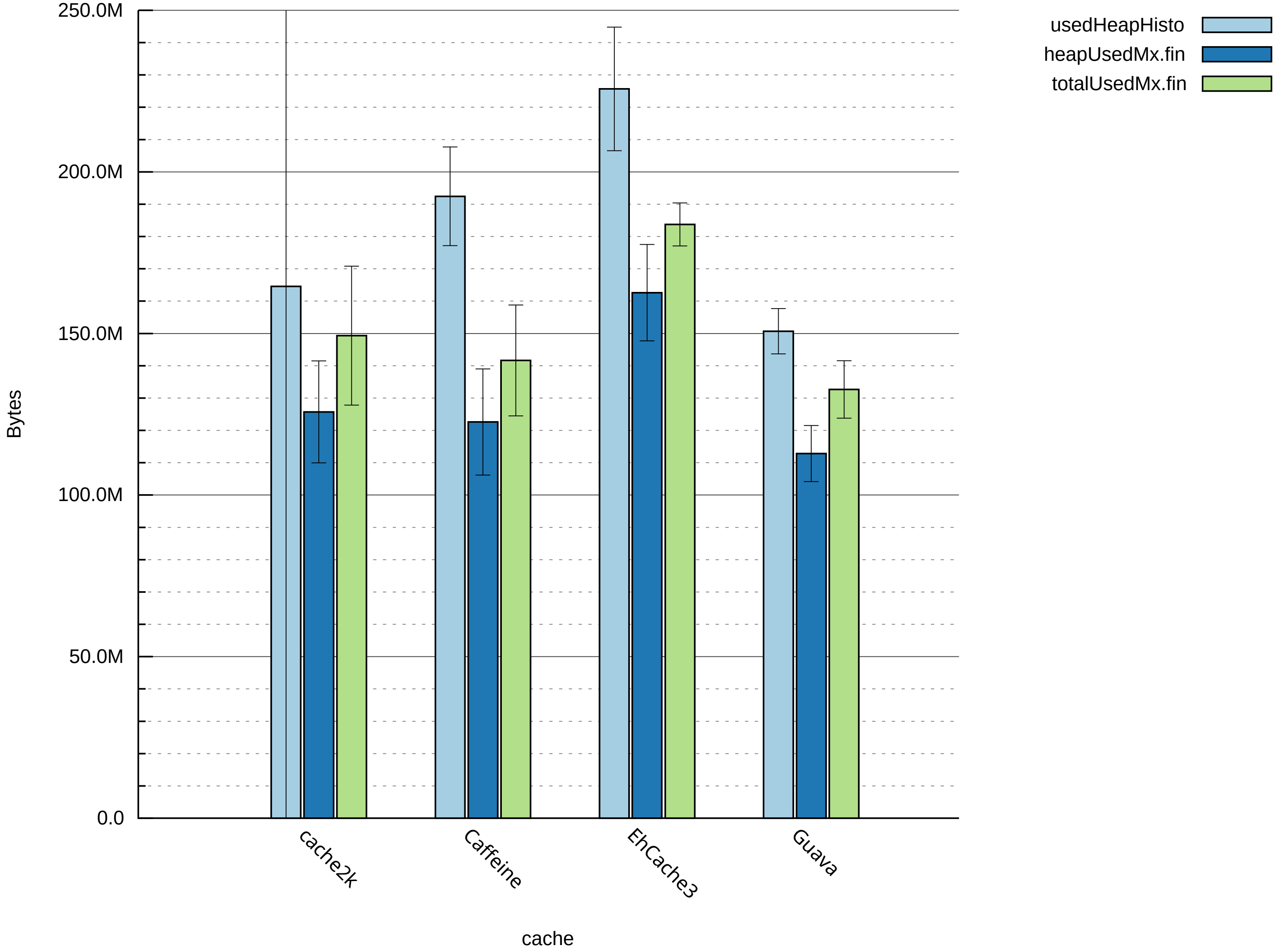
- JMH Parameters:
  - 3 forks, 2 warmup iterations, 3 measurement iterations, 60 seconds iterations time

  => 9 measurement iterations
- Graphs show the confidence interval
- Confidence interval is at 99.9% confidence level!

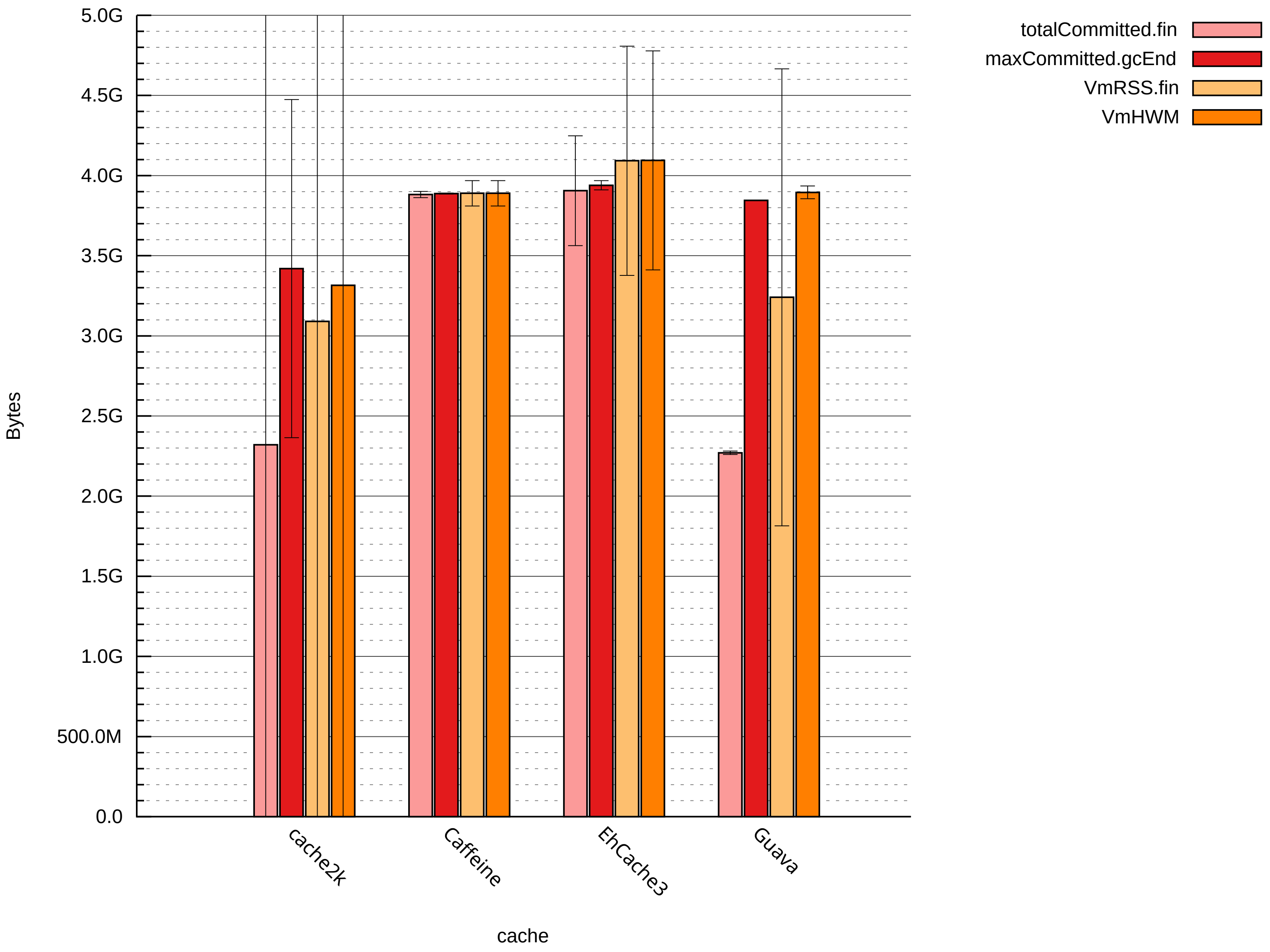
Parllel GC: ops/s



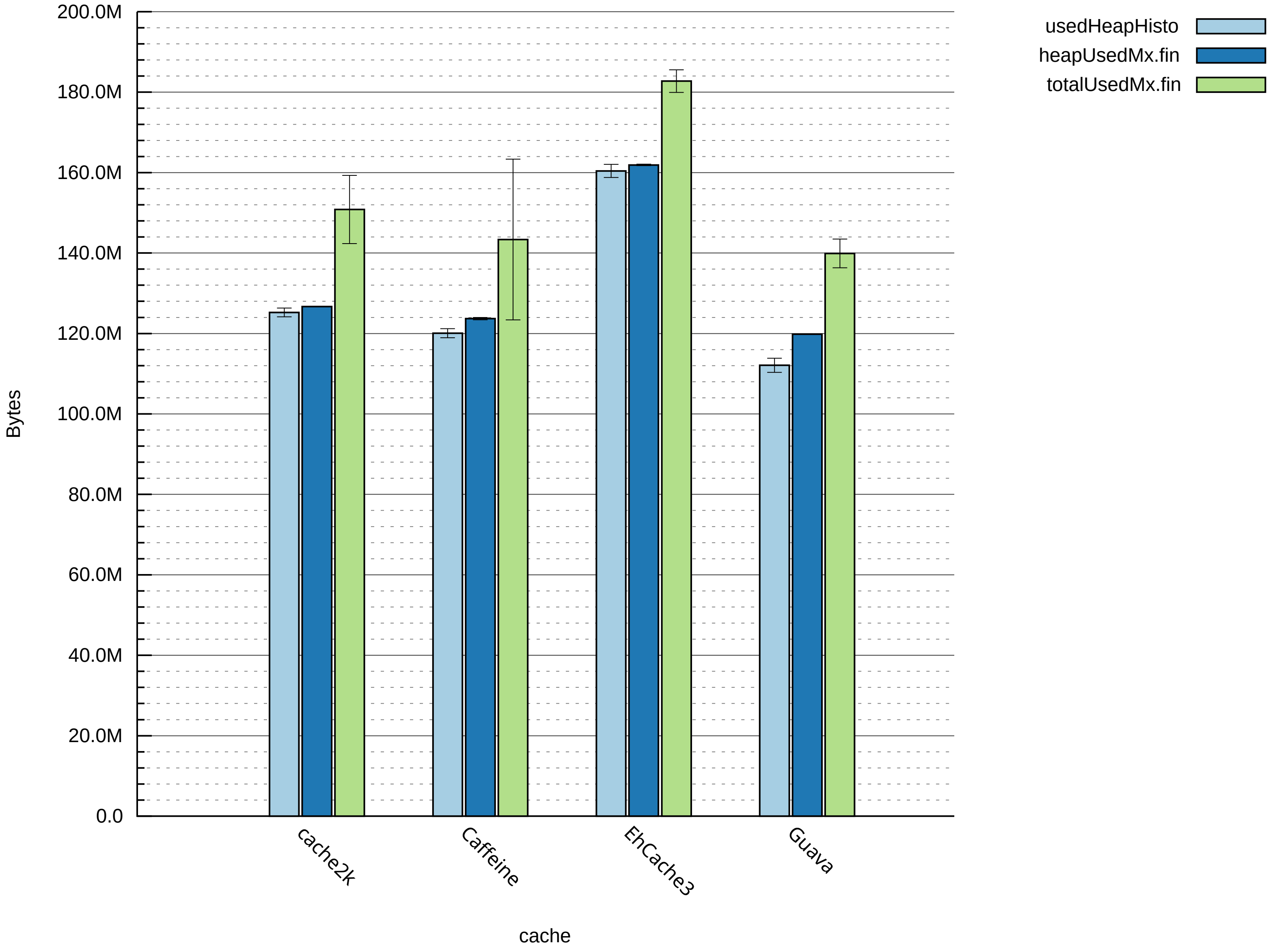
Parallel GC: heap usage



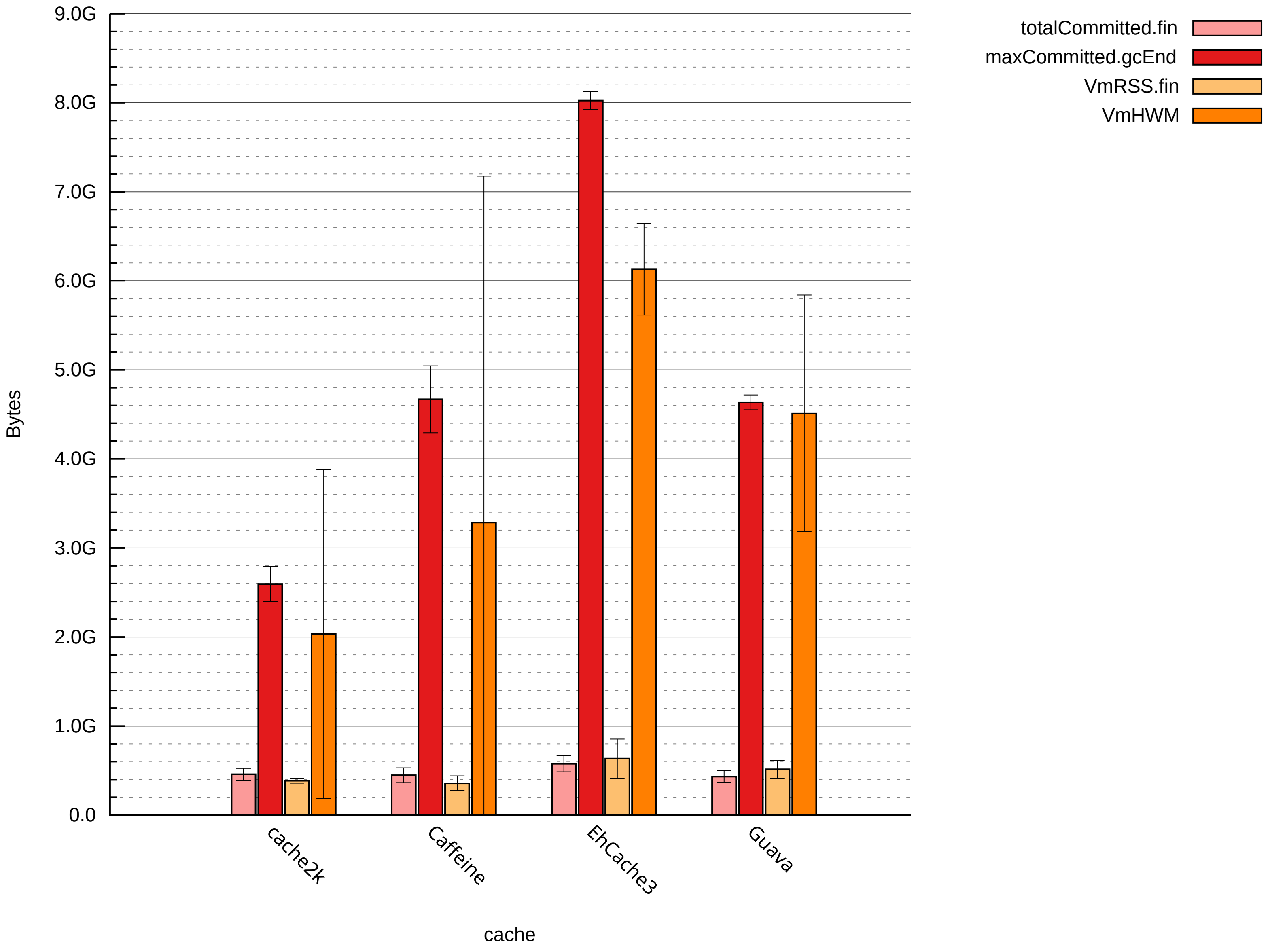
Parallel GC: Total / RSS



G1: heap usage

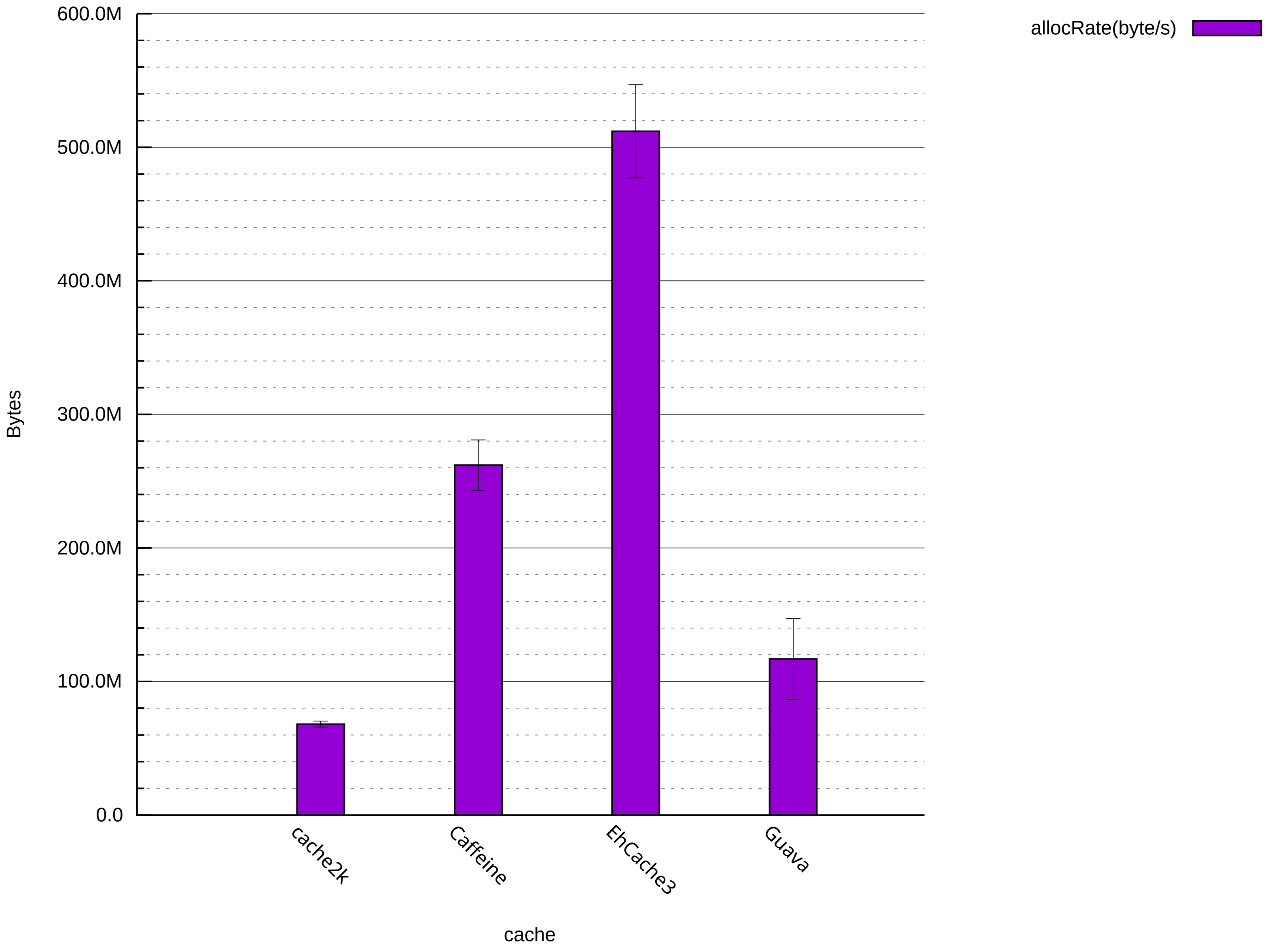


G1: Total / RSS

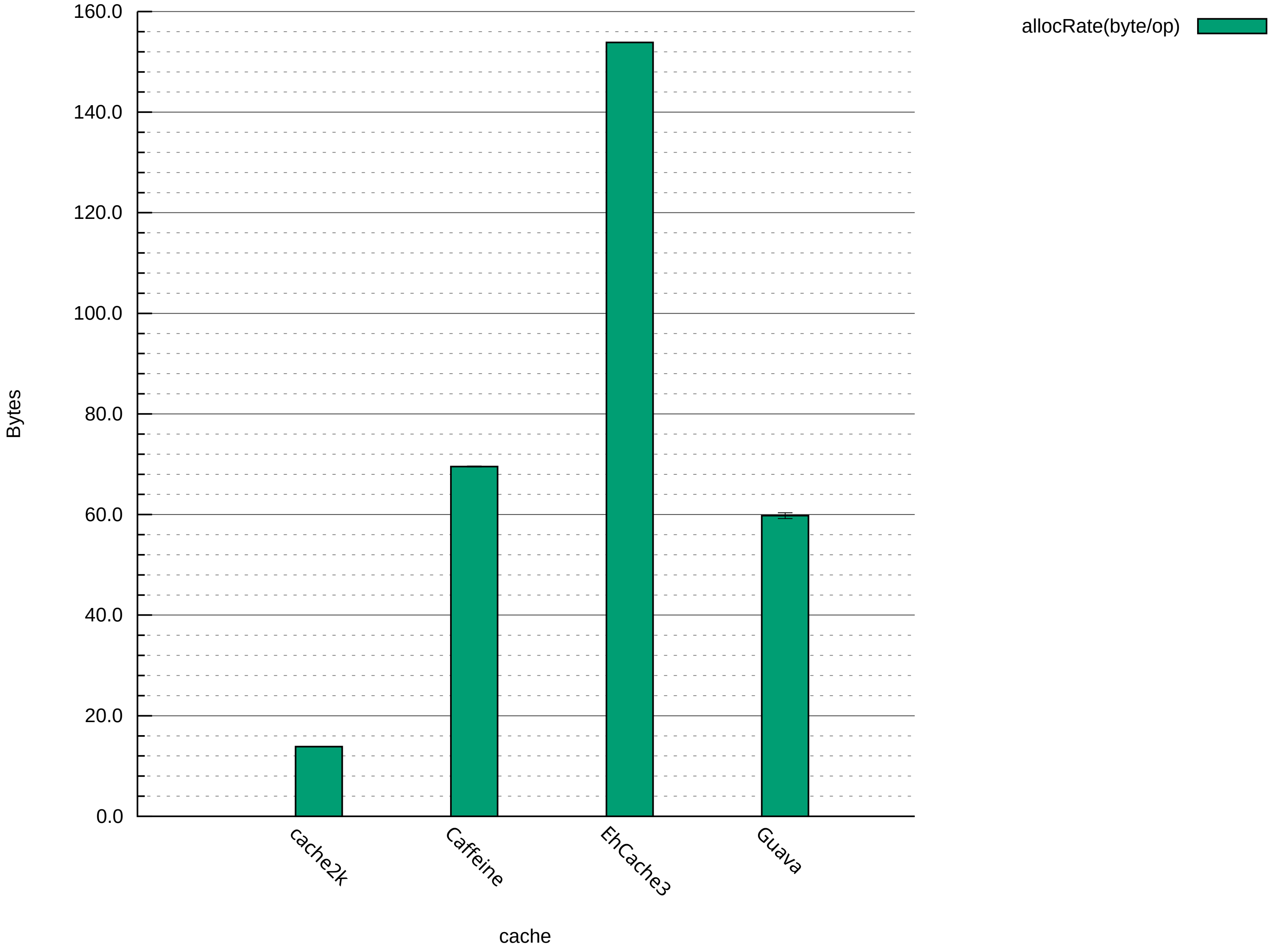


totalCommitted.fin
maxCommitted.gcEnd
VmRSS.fin
VmHWM
Bytes
9.0G
8.0G
7.0G
6.0G
5.0G
4.0G
3.0G
2.0G
1.0G
0.0
cache2k
Caffeine
EhCache3
Guava
cache

G1: allocation rate op/s



G1: allocation rate byte/op



Conclusion

- If GC is happening, also record memory usage in your JMH results
- JVM / OS memory metrics can be utilized, with varying degree of truth and consistency
- If GC is happening, your „micro“benchmarks need to run longer => Bigger heap, longer running time
- JMH can also be used to construct benchmarks to evaluate (peak) memory consumptions
- Be aware of the different GC implementations and behaviors

Thanks & Enjoy Live!

- JMH extensions available at: `github.com/cache2k/cache2k-benchmark`
  - `ForcedGcMemoryProfiler`
  - `LinuxVmProfiler`
  - Allocation rate via: `-prof gc`
- Plan/idea: discuss and add it into the JMH code base
- Please like:

  `github.com/cache2k/cache2k`

Notes

- How to start visualvm with bigger font size to make screenshots:

  ```
  visualvm -J-Dsun.java2d.uiScale=1.0 --fontsize 18
  ```

- First VisualVm heap graph from:

  ```
  java-11 -jar jmh-suite/target/benchmarks.jar \\.ZipfianSequenceLoadingBenchmark -jvmArgs -server\ -Xmx10G\ -XX:BiasedLockingStartupDelay=0\ -verbose:gc\ -XX:+UseParallelGC -f 1 -wi -w 15s -i 3 -r 15s -prof comp -prof gc -prof hs_rt -prof hs_gc -prof org.cache2k.benchmark.jmh.LinuxVmProfiler -prof org.cache2k.benchmark.jmh.MiscResultRecorderProfiler -prof org.cache2k.benchmark.jmh.GcProfiler -t 4 -p factor=5 -p entryCount=1000000 -p cacheFactory=org.cache2k.benchmark.Cache2kFactory -rf json -rff /run/shm/jmh-result/result-Cache2kFactory-ZipfianSequenceLoadingBenchmark-4.json
  ```

- Second VisualVm heap graph, additional parameters:

  ```
  -prof org.cache2k.benchmark.jmh.ForcedGcMemoryProfiler -gc true
  ```