Benchmarking/Profiling (In)sanity

It all started when I stumbled upon AppendableWriter in guava which is nothing more than an "adapter" class that adapts an Appendable to a Writer:

```
// It turns out that creating a new String is usually as fast, or faster
// than wrapping cbuf in a light-weight CharSequence.
target.append(new String(cbuf, off, len));
```

This statement smells… it looks like a decision made to quickly based on either a bad benchmark or a bad interpretation of the benchmark results. (which is unfortunately the norm)
So I created an implementation with a light weight CharSequence, with a comment claiming the opposite (with slightly more detail though):

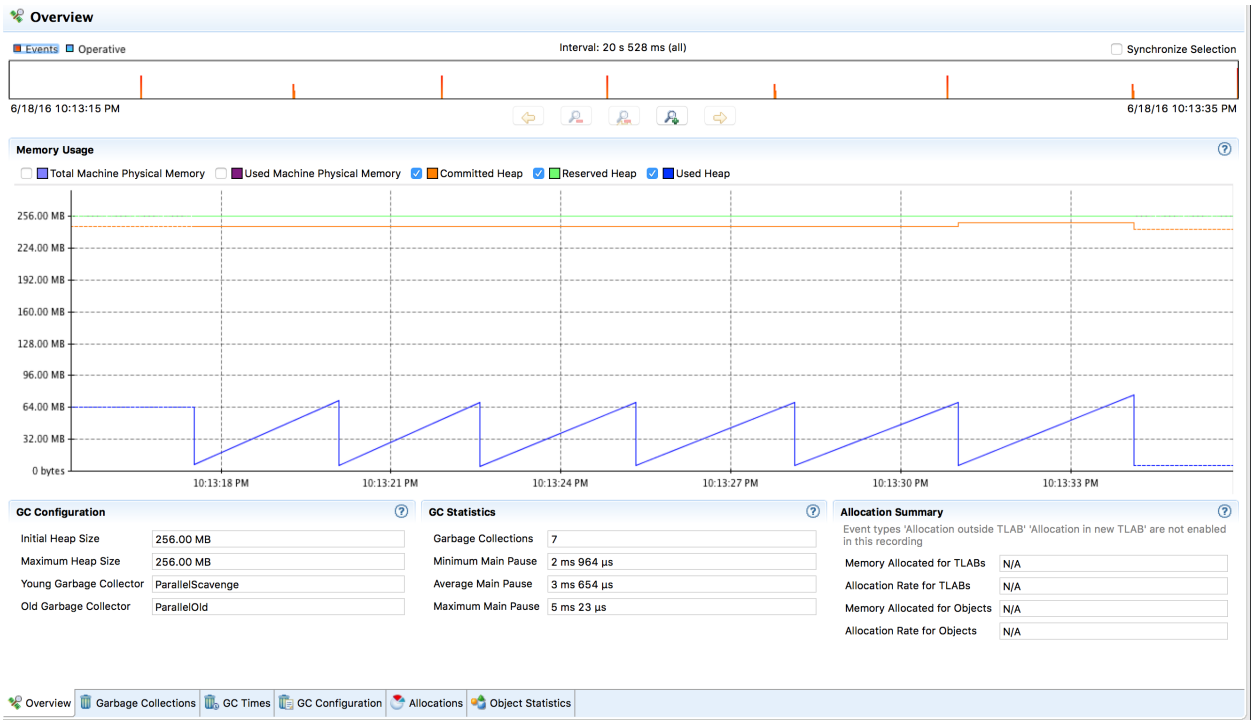
```
/*
  I wrote a JMH benchmark, and the results are as expected the opposite:

  Benchmark                                  Mode  Cnt      Score     Error  Units
  AppendableWriterBenchmark.guavaAppendable    thrpt   10  10731.940 ± 427.258  ops/s
  AppendableWriterBenchmark.spf4jAppendable    thrpt   10  17613.093 ± 344.769  ops/s

   Using a light weight wrapper is more than 50% faster!
   See AppendableWriterBenchmark for more detail...
  */
  appendable.append(CharBuffer.wrap(cbuf), off, off + len);
```

I have written a benchmark to validate my results based on the JMH benchmarking framework… (complete code at). I run these benchmarks s part of the release process of spf4j. The world was good for a while... after several JVM updates, OS updates.... I noticed something weird about my benchmark:

```
# JMH 1.12 (released 75 days ago)
# VM version: JDK 1.8.0_92, VM 25.92-b14
# VM invoker: /Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/jre/bin/java
# VM options: -Xmx256m -Xms256m -XX:+UnlockCommercialFeatures -Djmh.stack.profiles=/Users/zoly/NetBeansProjects/spf4j/spf4j-
benchmarks/target -Dspf4j.executors.defaultExecutor.daemon=true -Djmh.executor=FJP -Djmh.fr.options=defaultrecording=true,settings=/Users/
zoly/NetBeansProjects/spf4j/spf4j-benchmarks/src/main/jfc/profile.jfc
# Warmup: 10 iterations, 1 s each
# Measurement: 10 iterations, 1 s each
# Timeout: 10 min per iteration
# Threads: 8 threads, will synchronize iterations
# Benchmark mode: Throughput, ops/time
```

```
# Benchmark: com.google.common.io.AppendableWriterBenchmark.spf4jAppendable
# Run progress: 50.00% complete, ETA 00:00:21
# Fork: 1 of 1
# Preparing profilers: JmhFlightRecorderProfiler
# Profilers consume stderr from target VM, use -v EXTRA to copy to console
# Warmup Iteration   1: 70485.272 ops/s
# Warmup Iteration   2: 77614.082 ops/s
# Warmup Iteration   3: 4704.416 ops/s
# Warmup Iteration   4: 4730.839 ops/s
# Warmup Iteration   5: 4722.175 ops/s
# Warmup Iteration   6: 4718.885 ops/s
# Warmup Iteration   7: 4575.629 ops/s
# Warmup Iteration   8: 4710.284 ops/s
# Warmup Iteration   9: 4805.996 ops/s
# Warmup Iteration  10: 4747.728 ops/s
Iteration   1: 4903.879 ops/s
Iteration   2: 4912.047 ops/s
Iteration   3: 4972.782 ops/s
Iteration   4: 5015.564 ops/s
Iteration   5: 5021.001 ops/s
Iteration   6: 4958.149 ops/s
Iteration   7: 4883.546 ops/s
Iteration   8: 4733.823 ops/s
Iteration   9: 4800.601 ops/s
Iteration  10: 4896.969 ops/s
```

10x degradation during warmup ?!!! What the heck? This smells like a memory issue... but what the **** is going on? There are no leaks as the memory chart captured during the benchmark looks benign:

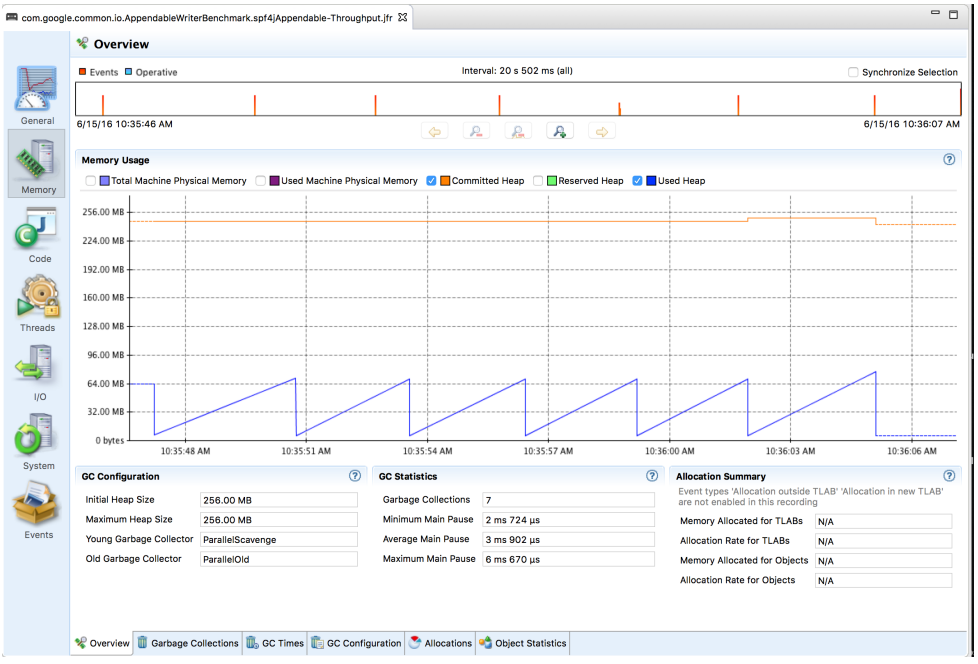


I can see that GC is working smoothly, why would some piece of code just slow down? (warmup is supposed to make things faster after all…), is the JIT doing a “deoptimization”?, is it memory

allocations? Lets use the “drunk man anti-method” (see for detail) and do an experiment with G1GC:

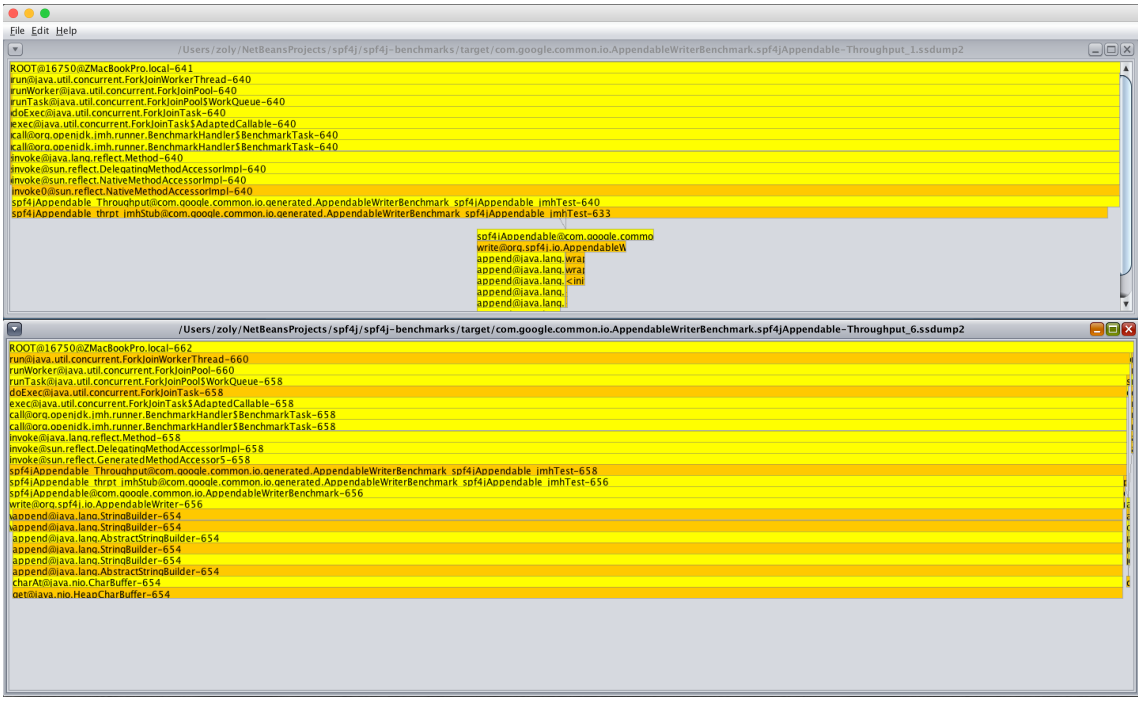
```
# JMH 1.12 (released 75 days ago)
# VM version: JDK 1.8.0_92, VM 25.92-b14
# VM invoker: /Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/jre/bin/java
# VM options: -XX:+UseG1GC -Xmx256m -Xms256m -XX:+UnlockCommercialFeatures -Djmh.stack.profiles=/Users/zoly/NetBeansProjects/
spf4j/spf4j-benchmarks/target -Dspf4j.executors.defaultExecutor.daemon=true -Djmh.executor=FJP -
Djmh.fr.options=defaultrecording=true,settings=/Users/zoly/NetBeansProjects/spf4j/spf4j-benchmarks/src/main/jfc/profile.jfc
# Warmup: 10 iterations, 1 s each
# Measurement: 10 iterations, 1 s each
# Timeout: 10 min per iteration
# Threads: 8 threads, will synchronize iterations
# Benchmark mode: Throughput, ops/time
# Benchmark: com.google.common.io.AppendableWriterBenchmark.spf4jAppendable

# Run progress: 50.00% complete, ETA 00:00:21
# Fork: 1 of 1
# Preparing profilers: JmhFlightRecorderProfiler
# Profilers consume stderr from target VM, use -v EXTRA to copy to console
# Warmup Iteration   1: 62995.070 ops/s
# Warmup Iteration   2: 70660.918 ops/s
# Warmup Iteration   3: 73127.240 ops/s
# Warmup Iteration   4: 69744.209 ops/s
# Warmup Iteration   5: 71235.279 ops/s
# Warmup Iteration   6: 70993.784 ops/s
# Warmup Iteration   7: 73375.331 ops/s
# Warmup Iteration   8: 74783.664 ops/s
# Warmup Iteration   9: 74197.163 ops/s
# Warmup Iteration  10: 74138.756 ops/s
Iteration   1: 73277.416 ops/s
Iteration   2: 72664.450 ops/s
Iteration   3: 72385.558 ops/s
Iteration   4: 71675.424 ops/s
Iteration   5: 74182.841 ops/s
Iteration   6: 75584.596 ops/s
Iteration   7: 74036.789 ops/s
Iteration   8: 72369.089 ops/s
Iteration   9: 72409.741 ops/s
Iteration  10: 72425.595 ops/s
```

Surprise surprise! The performance degradation magically disappears... heap chart looks good as well:

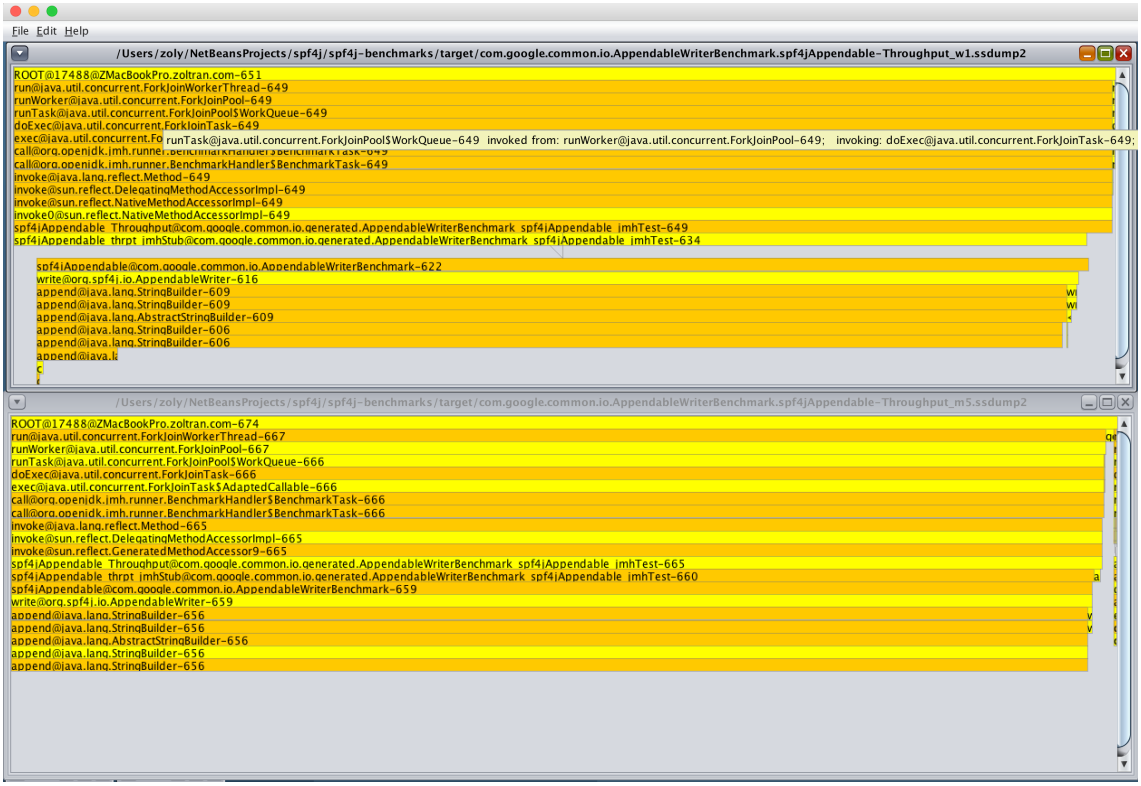


So for whatever reason this degradation does not happen with the G1 Garbage collector, but why? I can't image G1 GC being so magic (there is no magic in software)…

I have improved the SP4j profiler (www.spf4j.org) to get the ability to collect iteration specific Stack samples with JMH (Spf4jJmhProfiler) and here is a comparison between Iteration 1 (fast) and Iteration 6 (slow) with the default GC (Parallel):



Here are 2 iteration views (one from warmup, another one from measurement) with G1GC:



Hmm in the G1 world, like in the observed fast iteration with the Parallel GC, we don't see the HeapCharBuffer.get invocation as much, which means this invocation is probably inlined…. Which makes me think that a de-optimization happens and this de-optimization is related somehow to Garbage collection?…

Lets dig in further, and turn on compilation output (-XX:+PrintCompilation):

```
  5697  654       3       java.nio.Buffer::position (43 bytes)   made zombie
  5697  662       3       java.nio.Buffer::limit (62 bytes)   made zombie
  5697  969       3       org.apache.avro.generic.GenericData::getField (11 bytes)
  5697  741       3       java.nio.HeapCharBuffer::get (15 bytes)   made zombie
  5697  739       3       java.nio.CharBuffer::charAt (16 bytes)   made zombie
  5697  745       3       java.nio.Buffer::<init> (121 bytes)   made zombie
  5697  747       3       java.nio.CharBuffer::<init> (22 bytes)   made zombie
  5697  754       3       java.lang.StringBuilder::append (8 bytes)   made zombie
  5697  750       3       71901.755 ops/s — LAST time it was fast!
java.nio.CharBuffer::length (5 bytes)   made zombie
  5697  753       3       org.spf4j.io.AppendableWriter::write (15 bytes)   made zombie
  5697  757       3       java.lang.AbstractStringBuilder::append (54 bytes)   made zombie
  5697  755       3       java.lang.StringBuilder::append (6 bytes)   made zombie
  5697  749       3       java.lang.AbstractStringBuilder::append (144 bytes)   made zombie
  5699  971       3       java.lang.String::toString (2 bytes)
  5700  972       3       java.io.ObjectOutputStream$BlockDataOutputStream::setBlockDataMode (35 bytes)
# Warmup Iteration   3:   5728  973       3       java.util.Arrays::fill (21 bytes)
  6712  975       3       java.util.ArrayList::<init> (12 bytes)
  8121  984     n 0       java.lang.Class::isInstance (native)
  8122  985       3       java.io.ObjectOutputStream::writeHandle (21 bytes)
5461.899 ops/s]
```

They key methods of the Benchmark seem to get transformed into zombies :-) (which is never good, everybody knows that zombies will come at you in a clumsy way and will do bad things to you). We are getting closer now… I seems like a de-optimization happens… let's dig in and try to understand why (good explanation worth reading at https://gist.github.com/chrisvest/2932907).
Since the difference in behavior makes this smelling like a memory related issue, lets look at the usage of the code cache (-XX:+PrintCodeCache):

```
CodeCache: size=245760Kb used=3366Kb max_used=3366Kb free=242393Kb
bounds [0x000000010e997000, 0x000000010ed07000, 0x000000011d997000]
total_blobs=1280 nmethods=897 adapters=297
compilation: enabled
```

There seems to be no problem there… memory is not the issue here…
Lets look next into inlining(-XX:+PrintInlining):

```
                @ 47   org.spf4j.io.AppendableWriter::close (39 bytes)   inline (hot)
                  @ 8   org.spf4j.io.AppendableWriter::flush (24 bytes)   inline (hot)
                    @ 1   org.spf4j.io.AppendableWriter::checkNotClosed (35 bytes)   inline (hot)
                @ 23   org.openjdk.jmh.infra.Blackhole::consume (42 bytes)   disallowed by CompilerOracle
                @ 20   com.google.common.io.AppendableWriterBenchmark::spf4jAppendable (52 bytes)   force inline by CompilerOracle
                  @ 7   java.lang.StringBuilder::setLength (6 bytes)   inline (hot)
                    @ 2   java.lang.AbstractStringBuilder::setLength (45 bytes)   inline (hot)
```

```
                    @ 15   java.lang.AbstractStringBuilder::ensureCapacityInternal (16 bytes)   inline (hot)
                  @ 15   org.spf4j.io.AppendableWriter::<init> (23 bytes)   inline (hot)
                   @ 1   java.io.Writer::<init> (10 bytes)   inline (hot)
                    @ 1   java.lang.Object::<init> (1 bytes)   inline (hot)
                  @ 36   org.spf4j.io.AppendableWriter::write (15 bytes)   inline (hot)
                   @ 5   java.nio.CharBuffer::wrap (8 bytes)   inline (hot)
!                    @ 4   java.nio.CharBuffer::wrap (20 bytes)   inline (hot)
                     @ 7   java.nio.HeapCharBuffer::<init> (14 bytes)   inline (hot)
                      @ 10   java.nio.CharBuffer::<init> (22 bytes)   inline (hot)
                       @ 6   java.nio.Buffer::<init> (121 bytes)   inline (hot)
                        @ 1   java.lang.Object::<init> (1 bytes)   inline (hot)
                        @ 55   java.nio.Buffer::limit (62 bytes)   inline (hot)
                        @ 61   java.nio.Buffer::position (43 bytes)   inline (hot)
                   @ 8   java.lang.StringBuilder::append (6 bytes)   inline (hot)
                   \-> TypeProfile (307072/307072 counts) = java/lang/StringBuilder
                   @ 2   java.lang.StringBuilder::append (8 bytes)   inline (hot)
                    @ 2   java.lang.AbstractStringBuilder::append (54 bytes)   inline (hot)
                     @ 45   java.nio.CharBuffer::length (5 bytes)   inline (hot)
                      @ 1   java.nio.Buffer::remaining (10 bytes)   inline (hot)
                     @ 50   java.lang.StringBuilder::append (8 bytes)   inline (hot)
                      @ 4   java.lang.StringBuilder::append (10 bytes)   inline (hot)
                       @ 4   java.lang.AbstractStringBuilder::append (144 bytes)   inline (hot)
                        @ 18   java.nio.CharBuffer::length (5 bytes)   inline (hot)
                         @ 1   java.nio.Buffer::remaining (10 bytes)   inline (hot)
                        @ 89   java.lang.AbstractStringBuilder::ensureCapacityInternal (16 bytes)   inline (hot)
                         @ 12   java.lang.AbstractStringBuilder::expandCapacity (50 bytes)   inline (hot)
                          @ 43   java.util.Arrays::copyOf (19 bytes)   inlining too deep
                        @ 116   java.nio.CharBuffer::charAt (16 bytes)   inline (hot)
                         @ 2   java.nio.Buffer::position (5 bytes)   accessor
                         @ 8   java.nio.Buffer::checkIndex (24 bytes)   inline (hot)
                         @ 12   java.nio.HeapCharBuffer::get (15 bytes)   inline (hot)
                          @ 7   java.nio.Buffer::checkIndex (22 bytes)   inlining too deep
                          @ 10   java.nio.HeapCharBuffer::ix (7 bytes)   inlining too deep
```

Now that is something interesting! Inlining to deep?! and exactly for the method I am seeing in my profile when things are being slow!
Let's increase the inlining level a bit and see what happens (default is 9) (-XX:MaxInlineLevel=12) :

```
# Run progress: 0.00% complete, ETA 00:00:20
# Fork: 1 of 1
# Warmup Iteration   1: 83142.216 ops/s
# Warmup Iteration   2: 103245.276 ops/s
# Warmup Iteration   3: 96166.461 ops/s
# Warmup Iteration   4: 96081.599 ops/s
# Warmup Iteration   5: 95880.021 ops/s
# Warmup Iteration   6: 95450.397 ops/s
# Warmup Iteration   7: 94175.079 ops/s
# Warmup Iteration   8: 80373.298 ops/s
# Warmup Iteration   9: 76777.619 ops/s
# Warmup Iteration  10: 89284.943 ops/s
Iteration   1: 83158.890 ops/s
Iteration   2: 81364.498 ops/s
Iteration   3: 86231.231 ops/s
Iteration   4: 84652.819 ops/s
Iteration   5: 84800.907 ops/s
Iteration   6: 82218.109 ops/s
Iteration   7: 86128.886 ops/s
Iteration   8: 84813.081 ops/s
Iteration   9: 79925.358 ops/s
Iteration  10: 85968.824 ops/s


Benchmark                                Mode  Cnt      Score      Error  Units
AppendableWriterBenchmark.spf4jAppendable  thrpt   10  83926.260 ± 3285.072  ops/s
```

Hurray! No more degradation with the Parallel GC!!!! There is still something interesting happening in warmup iteration 2 though… (I have a feeling more can be squeezed out here…and it has to do with zombies :-))

Let's redo the benchmarks with our newly learned tweaks…

With G1GC: # VM options: -XX:MaxInlineLevel=12 -XX:+UseG1GC -Xmx256m -Xms256m

Benchmark	Mode	Cnt	Score	Error	Units
AppendableWriterBenchmark.guavaAppendable	thrpt	10	44132.545	± 615.835	ops/s
AppendableWriterBenchmark.spf4jAppendable	thrpt	10	**76099.587**	± 1877.786	ops/s

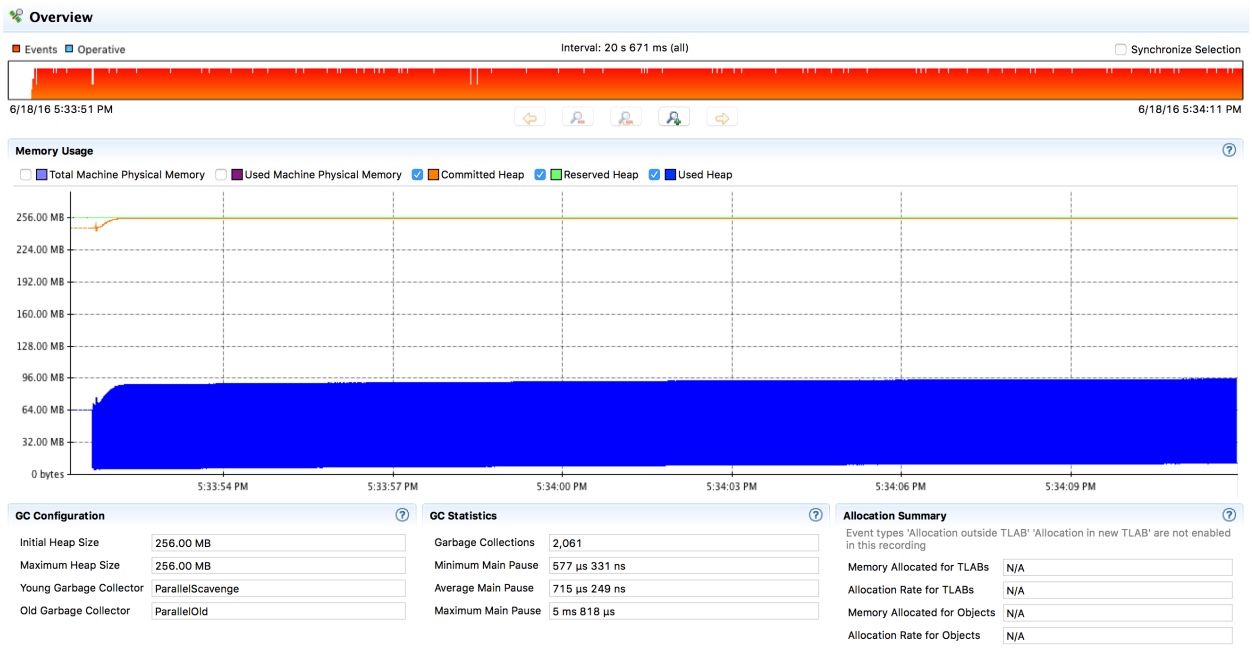
With Parallel GC(default in my case): # VM options:-XX:MaxInlineLevel=12 -Xmx256m -Xms256m

Benchmark	Mode	Cnt	Score	Error	Units
AppendableWriterBenchmark.guavaAppendable	thrpt	10	45104.009	± 395.868	ops/s
AppendableWriterBenchmark.spf4jAppendable	thrpt	10	**81934.901**	± 1121.964	ops/s

We observe significantly better performance with the spf4j implementation with both GC implementations. And this time G1 GC looses its magic…

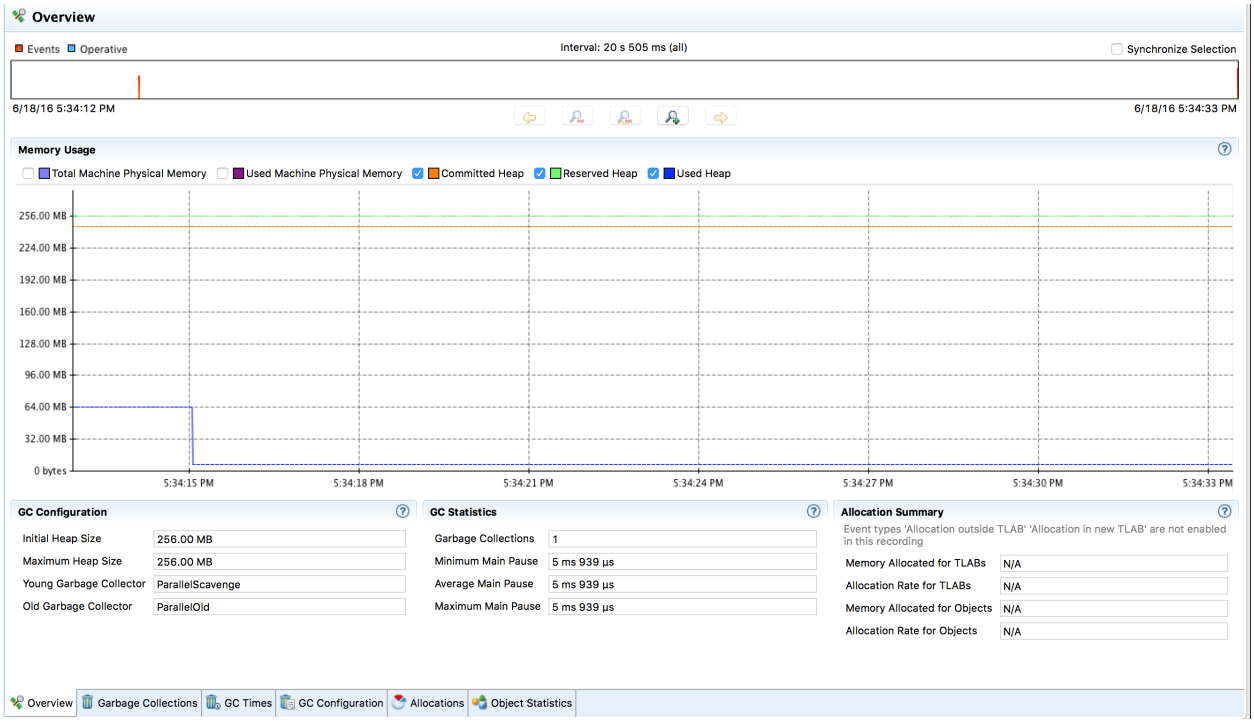
We could have made a naive conclusion that G1GC is superior to the Parallel GC based on one of my first observations (drunk man anti-method sometimes leads to drunk man conclusions)… Or the conclusion the guava developers made…

A few more data points to visualize the resource usage difference…
The guava implementation Memory/GC overview:



Where we can see lots of allocations and garbage collection being done really fast, making sure the CPU is not bored..

And the spf4j implementation Memory/GC overview:



A lot less garbage created and collected which is where the extra performance (2x) is coming from. The quietness we see is probably also related to some extra escape analysis which happens most likely to the char buffer involved here. It is always good to have the GC bored… Think of all the other garbage your app can now create :-)

Conclusion:

1) Benchmarking is hard. Really hard. Beware of benchmarks and benchmark results, most benchmarks and conclusions I am seeing on the web are BAD, be skeptical.
2) Disregard any benchmark not made with a good framework (like JMH, see for more detail)
3) Single threaded benchmarks are not valuable most of the time, resource hogs might perform well single threaded, but when multithreaded they are less likely to do so.
4) Disregard benchmark results/conclusions that lack detail.
5) Understanding benchmarking measurements is hard. Really hard. Do NOT jump to conclusions, try to understand what is going on.
6) Drunk man anti-method can lead to drunk man conclusions, but it can still provide interesting knowledge.
7) Beware of tools, measuring a system changes a system, they will impact your measurements it is good to have a good understanding of how they work, and what the potential impact might be. Validate measurements made with one tool with another tool. (One of the reason I am using 2 profilers)

The tools used in this analysis are:

1) Java Flight Recorder + Java Mission Control to record and visualized profiles of the benchmarks. (http://www.oracle.com/technetwork/java/javaseproducts/mission-control/java-mission-control-1998576.html)
2) Spf4j stack sampling profiler + UI to record and visualize profiles. (http://www.spf4j.org)
3) JMH benchmarking framework. (http://openjdk.java.net/projects/code-tools/jmh/)

I made all the efforts to make the measurements and conclusions in this article as “correct” as possible, but they are not perfect, more measurements on more platforms would have been nice, with more iterations, on server hardware (not a macbook pro… although with 4 cores and 16Gb of ram + SSD it makes a better server than most of the cloud servers out there :-))…
The benchmark should also be improved to make it more realistic to actual real life workloads….

One thing we still have unanswered is: why is the inlining not an issue with G1 GC?
I will leave this for another day…

All code is on github: https://github.com/zolyfarkas/spf4j/tree/master/spf4j-benchmarks
For any comments/questions, email me at: zolyfarkas@yahoo.com

Enjoy!