String Density: Performance and Footprint

Aleksey Shipilev
Oracle
aleksey.shipilev@oracle.com

Xueming Shen
Oracle
xueming.shen@oracle.com

Brent Christian
Oracle
brent.christian@oracle.com

Charlie Hunt
Oracle
charlie.hunt@oracle.com

December 2, 2014

Contents

Chapter 1

Introduction

1.1 Background

The need for better memory footprint for Java applications is widely recognized. Over the years, we did quite a few optimizations with regards to memory footprint, and now we suggest to revisit the compressed Strings proposal [1]. Since this change takes on the very core of the platform class library, we have to exercise the utmost caution with justifying the need for the change, making the changes backed by the performance data, and accurately track if the actual improvements are on track with the predicted ones.

1.2 Caveats

THIS IS A WORK IN PROGRESS. We can not stress this enough. We mostly understand the footprint side of the story, but the throughput and latency impact on String APIs is still not clear. The goal for this performance work is to quantify if we can go forward with either improvement in production JDK code.

1.3 Summary

We have tried a relevant approach in JDK 6, but the maintenance costs were very high, and the performance impact was prohibitive in the face of converting from/to two different String shapes. A brief summary of current exploration is available in JEP proposal [1]. The relevant performance plan this report is covering is available as JEP task [2].

The key idea for new exploration is to optimize for the special case: Strings that have all characters represented by a single byte, not two bytes. Here and afterwards, we call these Strings *compressible*, as opposed to *non-compressible* Strings. The distinction between compressible and non-compressible Strings is treated on Java level, with VM recognizing both String shapes, and optimizing for them.

Chapter 2

Memory Footprint

2.1 Experimental Setup

Unless otherwise noted, we are using this experimental setup:

- **Heap Dump collection**, representing a collection of over 950 heap dumps from a variety of different Oracle software applications using Java including Oracle Fusion Middleware applications and Oracle Fusion Applications.
- **Java Object Layout (JOL) tools** [3]. In order to quantify the improvements properly, we conduct the study on large corpus of heap dumps, and simulate the suggested improvements on them. See Appendix A for instructions how to run the footprint experiments yourselves.
- **Linux, x86_64**, running on 1x4x2 i7-4790K. Since the footprints are oblivious to OS flavor, we limit our study only to one OS/HW platform.
- **OpenJDK 9** [5]. The following JVM modes are emulated:
  - 32-bit data model,
  - 64-bit data model, compressed references disabled
  - 64-bit data model, compressed references enabled
  - 64-bit data model, compressed references enabled, 16-byte object alignement

  We run in different JVM modes to vary object alignments, object header sizes, and also to gather the actual object layout for additional verification.

2.2 Results

Before we dive head-first into implementing the proposals, let us quickly estimate the potential improvements. In order to have the realistic estimates, we process the collection of heap dumps with Java Object Layout tools. Note that the number of live `String`/`char[]` objects can only be estimated, because liveness is only accurate right after a full collection.

Please note this approach also does not answer the question what `String`/`char[]` objects are allocated transiently during the application execution, but we believe the footprint improvements demonstrated on persistent objects also apply to the transient ones.

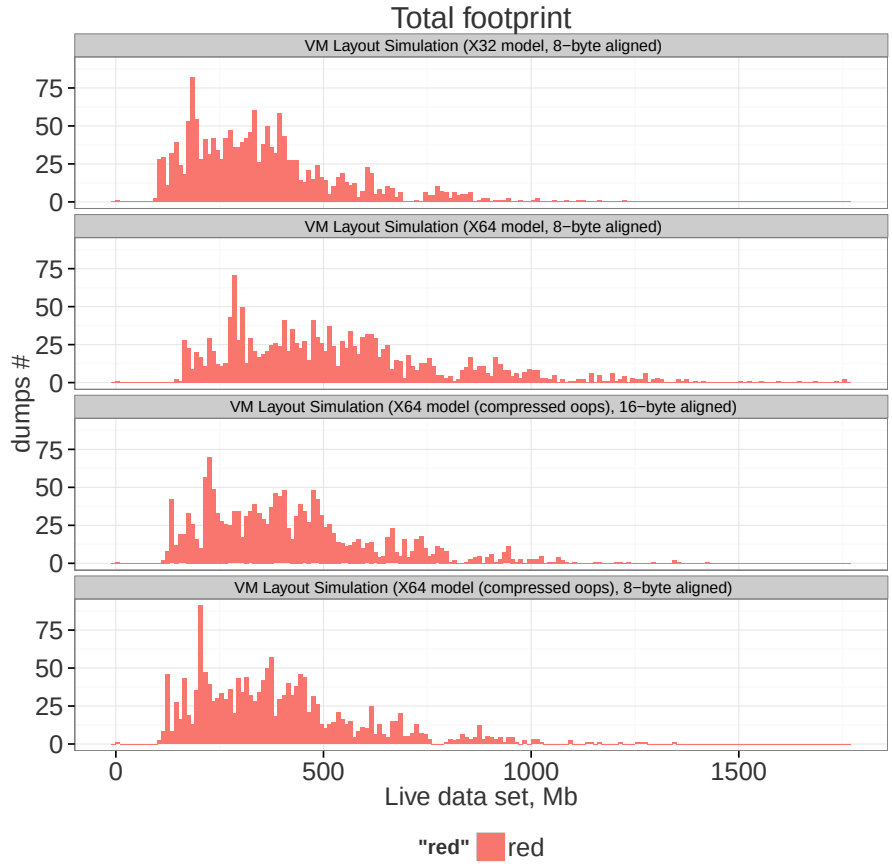

Figure 2.1: **Live dataset size in the heapdumps.**

Figure 2.1 shows the live data set (LDS) size distributions in those heap dumps. The footprint follows the same distribution in all heapdumps. The difference in reference size makes X64 result without the compressed references to balloon the footprint. The 16-byte alignment skews the distribution to larger sizes again.

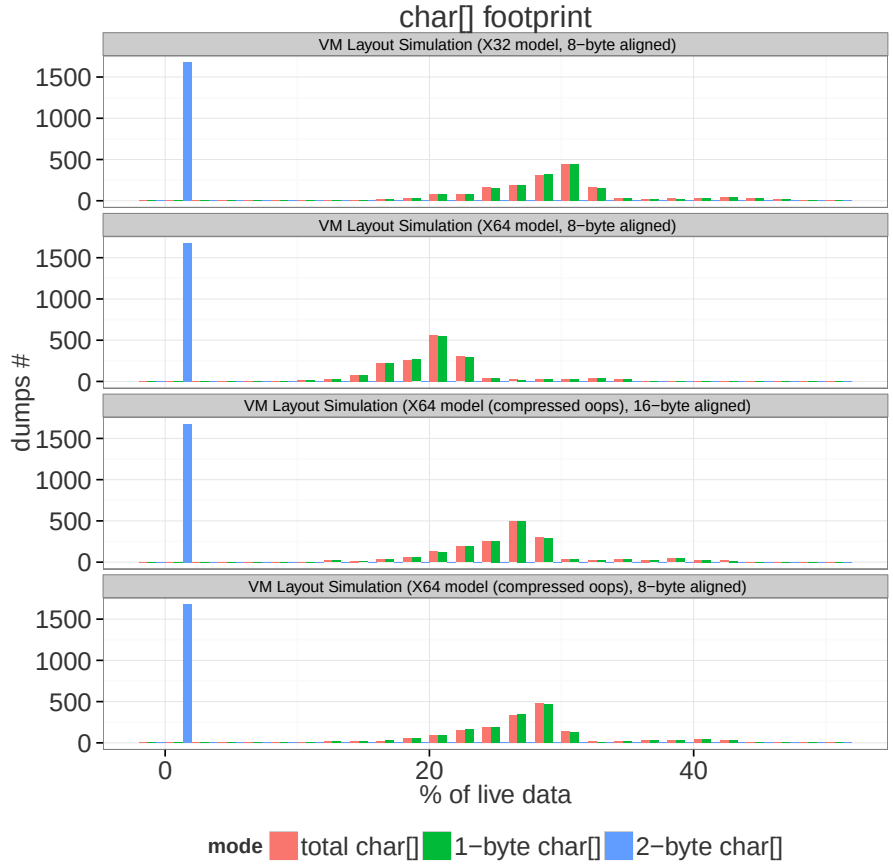

Figure 2.2: **char[] footprint.**

In order to test the assumption that character arrays consume a considerable amount of LDS, we tabulated the footprint consumed by `char[]` instances in the heap dumps. Figure 2.2 shows that character arrays consume a significant amount of LDS, between 10% and 45%, depending on concrete heap dump and VM mode. X64 mode without the compressed references has generally lower impact of `char[]`, because other larger size of other objects attenuate the character arrays footprint.

Notice that 1-byte characters vastly outnumber the 2-byte characters in almost all heap dumps. That gives us a reason to believe most of the `char[]` arrays are compressible.

```
***** 32-bit VM: ********************************************************
java.lang.String object internals:
 OFFSET  SIZE   TYPE DESCRIPTION                    VALUE
      0     8        (object header)                N/A
      8     4 char[] String.value                   N/A
     12     4    int String.hash                    N/A
Instance size: 16 bytes (estimated, the sample instance is not available)
Space losses: 0 bytes internal + 0 bytes external = 0 bytes total

***** 64-bit VM: ********************************************************
java.lang.String object internals:
 OFFSET  SIZE   TYPE DESCRIPTION                    VALUE
      0    16        (object header)                N/A
     16     8 char[] String.value                   N/A
     24     4    int String.hash                    N/A
     28     4        (loss due to the next object alignment)
Instance size: 32 bytes (estimated, the sample instance is not available)
Space losses: 0 bytes internal + 4 bytes external = 4 bytes total

***** 64-bit VM, compressed references enabled: **************************
java.lang.String object internals:
 OFFSET  SIZE   TYPE DESCRIPTION                    VALUE
      0    12        (object header)                N/A
     12     4 char[] String.value                   N/A
     16     4    int String.hash                    N/A
     20     4        (loss due to the next object alignment)
Instance size: 24 bytes (estimated, the sample instance is not available)
Space losses: 0 bytes internal + 4 bytes external = 4 bytes total

***** 64-bit VM, compressed references enabled, 16-byte align: ************
java.lang.String object internals:
 OFFSET  SIZE   TYPE DESCRIPTION                    VALUE
      0    12        (object header)                N/A
     12     4 char[] String.value                   N/A
     16     4    int String.hash                    N/A
     20    12        (loss due to the next object alignment)
Instance size: 32 bytes (estimated, the sample instance is not available)
Space losses: 0 bytes internal + 12 bytes external = 12 bytes total
```

Figure 2.3: **java.lang.String layout, simulated in different VM modes**

Proposed implementations add either a `boolean` or `reference` field to `String`. Luckily for us, there is a considerable amount of alignment shadow space in `String` to absorb the new field without making `String` larger. For example, Figure 2.2 shows that `java.lang.String` has 4 bytes wasted due to the 8-byte object alignment in almost all modes. This is why adding an additional `boolean` field or a compressed `reference` does not inflate the object size.

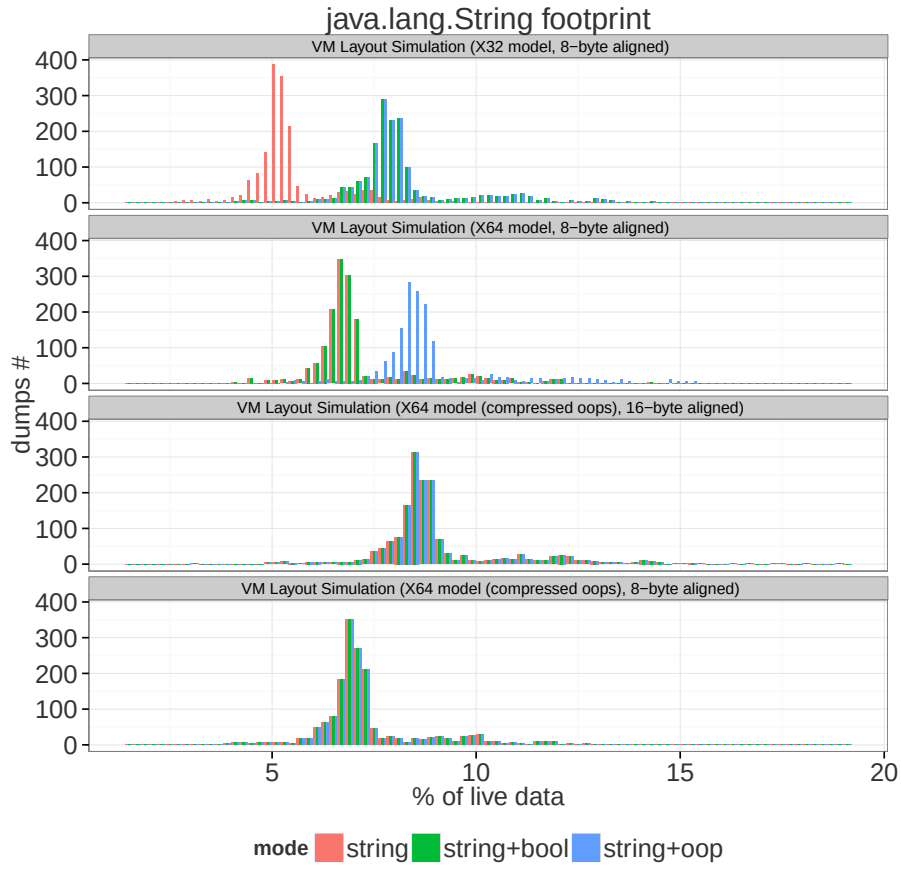

Figure 2.4: **String footprint**.

Figure 2.4 backs that observation up. The inflation only occurs on:

- 32-bit VM with additional boolean or reference field. There is no alignment shadow to cram the new field, and therefore we have to inflate `String` by 8 full bytes, despite the fact we only use a single byte, or four bytes there.
- 64-bit VM without compressed references, and a new reference is added. The alignment shadow can only absorb 4 bytes, and full 64-bit reference clearly breaks away from here, again, inflating `String` by 8 full bytes.

These overheads balloon the heap occupancy for `String`s considerably, around 2%-3% in the unlucky scenarios.

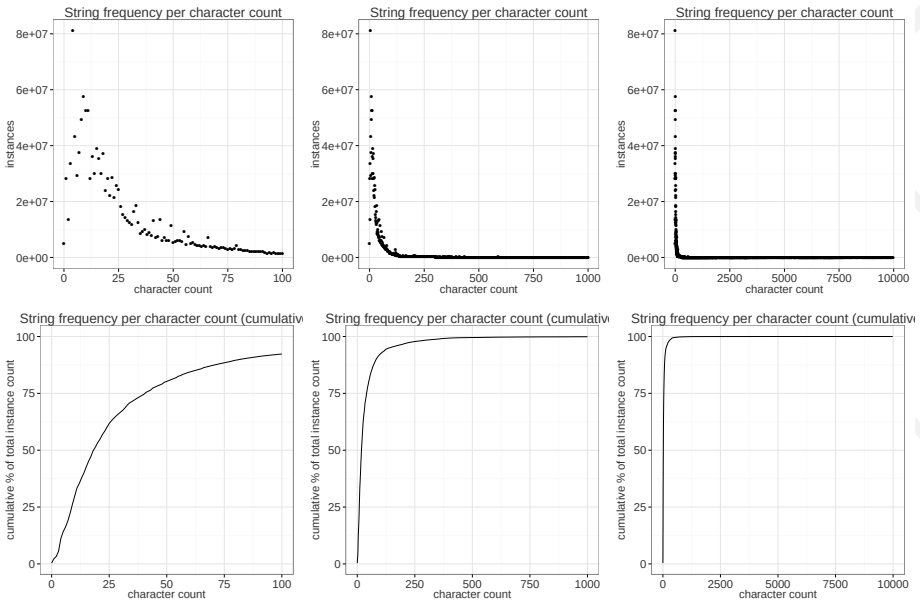

Figure 2.5: **String size distribution, 1**

Therefore, the potentential compression improvements are attenuated by the overheads in `java.lang.String`, `char[]` headers, and the alignment constraints. These overheads get lower as the number of characters per `String` grows, and so we need to see what `String`s are usually present in those heaps. Figure 2.5 highlights the vast majority of `String`s are rather small, with $> 75\%$ of all `String`s being smaller than 35 characters.

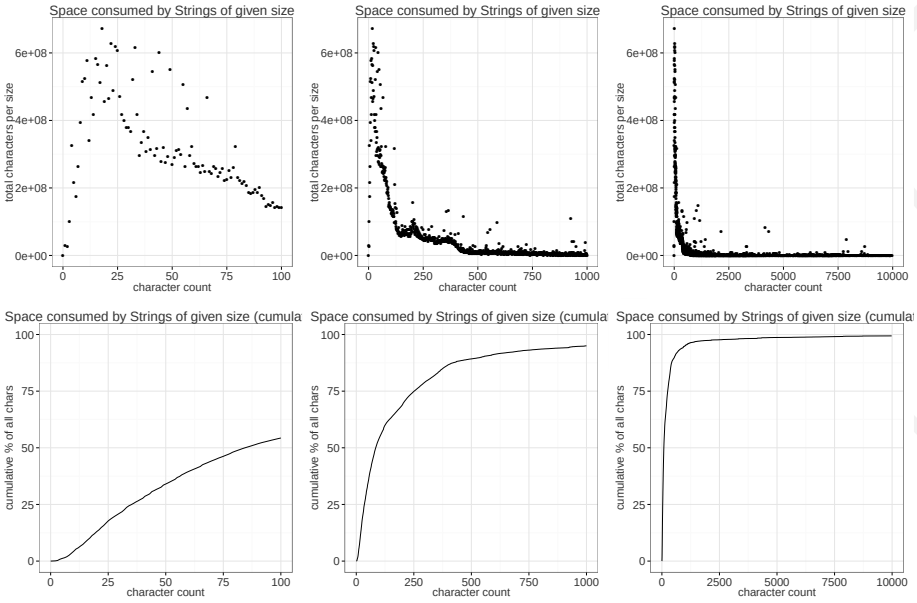

Figure 2.6: **String size distribution, 2**

Of course, the absolute memory savings depend not only on the `String` instance count, but rather by the total character count for the `String`s of particular size. Figure 2.6 shows that more occupied `String`s skew the footprint towards the tail. There, around 75% of all characters are residing in the `String`s of lengths less than 250 characters.

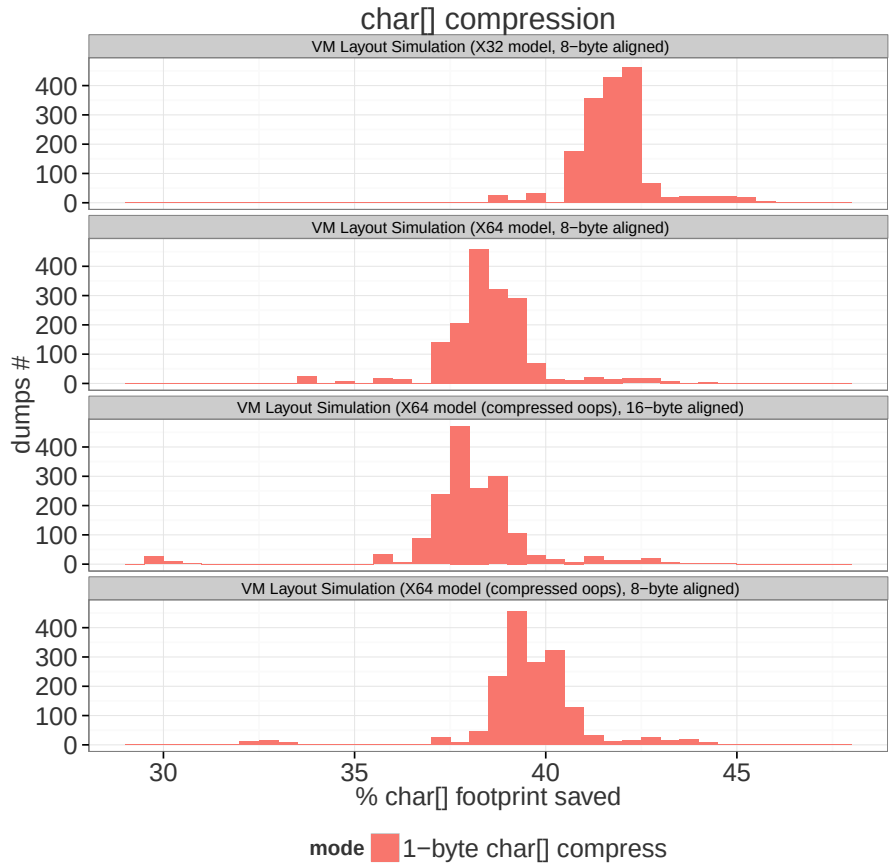

Figure 2.7: **char[] compressibility**

The high count of small `String`s may explain why the footprint improvement for compressing `String`s, while being large in absolute values, are being lower than theoretical 50% value. Figure 2.7 shows the projected `char[]` footprint improvements on these realistic heap dumps. Notice the improvements lie within 35%-40%.

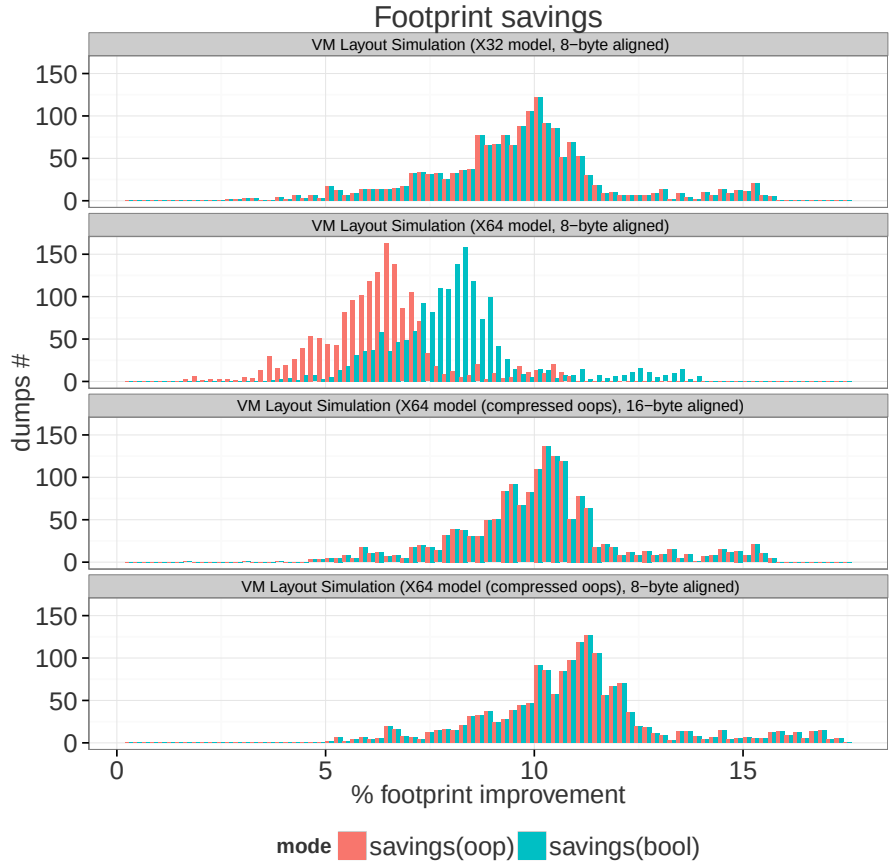

Figure 2.8: **Total footprint savings**

Now, if we simulate the total improvements for compressing `String`s, taking into the account the inflation of `String` objects, the compressibility of `char[]` arrays and the object overheads of them, we can arrive at Figure 2.8. It predicts the improvements within 5%-15% of the total heap occupancy.

Chapter 3

Throughput Performance

3.1 Experimental Setup

Unless otherwise noted, we are using this experimental setup:

- **Java Microbenchmark Harness (JMH)** [4]. The targeted microbenchmarks are done and analyzed with the help of JMH. The best microbenchmarking practices are employed under the guidance of Java SE Performance team. See Appendix B for instructions how to run the performance experiments yourself.
- **OpenJDK 9** [5]. The following JVM modes are emulated:
  - 32-bit data model,
  - 64-bit data model, compressed references disabled
  - 64-bit data model, compressed references enabled
  - 64-bit data model, compressed references enabled, 16-byte object alignement

  We run in different JVM modes to vary object alignments, object header sizes, and also to gather the actual object layout for additional verification.

3.2 Results

Under construction. We use the custom-built `String` benchmarks [6].

Chapter 4

Conclusions

4.1 Conclusion

Appendix A

Running JOL tools

Java Object Layout is distibuted in source form, and therefore it requires building. Please follow the instructions on JOL page [3] to build and run the JOL tools. For the sake of this report, we are using two tools from JOL toolbox:

1. **jol-string-compress**. This tool digests the heap dumps, and figures out `String` counts, `char[]` counts and their compressibility, and estimates the impact of two proposed improvements. It can be executed with:

   ```
   % java -jar jol-cli/target/jol-string-compress.jar *.hprof
   ```

   This invocation will digest the heap dumps passed via the command line, and emit a CSV output. This output can be rendered with the tools of your choice, as well as digested by humans.

   Running the same tool with `-Dmode=histo` will produce the `String` size distribution histogram:

   ```
   % java -Dmode=histo -jar jol-cli/target/jol-string-compress.jar *.hprof
   ```

2. **jol-estimates**. This tool *emulates* different JVM modes, and introspects the given class layout. The tool can be invoked via

   ```
   % java -jar jol-cli/target/jol-estimates.jar java.lang.String
   ```

   The example output can be seen in Figure 2.2. Note this output is not the actual layout, but only the simulation given on the educated guess how Hotspot lays out the object internals in different VM modes.

Appendix B

Running JMH-driven performance benchmarks

The benchmarks are under heavy development at this point, but you can build them yourself from source [6]. After you built the benchmarks, you may either run them via the usual JMH runner, using *benchmarks.jar*:

```
% java -jar target/benchmarks.jar
```

Or, you may use the composite runner, conveniently packed in *string-density-bench.jar*:

```
% java -jar target/string-density-bench.jar
```

Bibliography

[1] *More memory-efficient internal representation for Strings*,
`https://bugs.openjdk.java.net/browse/JDK-8054307`

[2] *Performance Plan for More memory-efficient internal representation for Strings*,
`https://bugs.openjdk.java.net/browse/JDK-8064810`

[3] *Java Object Layout (JOL) toolbox*,
`http://openjdk.java.net/projects/code-tools/jol/`

[4] *Java Microbenchmark Harness (JMH)*,
`http://openjdk.java.net/projects/code-tools/jmh/`

[5] *OpenJDK 9*,
`http://hg.openjdk.java.net/jdk9/jdk9/`

[6] *string-density-bench*,
`http://cr.openjdk.java.net/~shade/density/string-density-bench.zip`