Java Field Layouts: Qualitative Study on Maven Central

Aleksey Shipilev
Oracle Corporation
Moscow, Russia
aleksey.shipilev@oracle.com

ABSTRACT

OpenJDK Java Virtual Machine (HotSpot) is the reference implementation of JVM. The goals of the JVM are to provide a reliable, high-performance execution environment for Java programs.

One of the many duties of the JVMs is to allocate Java objects and manage object metadata. Users are normally oblivious to this, since there is little specified about the object layout and precise semantics with regards to memory layout in the Java Language Specification. This enabled the JVM to rearrange object fields to fit the platform requirements, including aligning the fields to evade misaligned accesses, packing the fields to reduce footprint, etc.

In this report, we quantify the JVM behavior with regards to field layout on a large corpus of class files. We show that while lots of classes benefit from the current layout scheme, the layout can still be significantly improved.

1. IMPORTANT INVARIANTS

Current HotSpot tries to rearrange fields to improve memory footprint, while still maintaining a few important invariants.

Field alignment.

In order to avoid misaligned accesses, we need to lay out the fields aligned by their size. For example, with 12-byte headers, we are not allowed to place an 8-byte *long* field at offset 12, because doing so will break the long alignment, which can result in losing read/write atomicity, spurious SIGBUSes, or invoking kernel fallback, thus code sacrificing performance. In this case, we can only put long at offset 16. The gaps resulting from maintaining this invariant contribute to *internal alignment losses*.

Object alignment.

In order for field alignment to work properly, and also to provide the atomicity while accessing the object headers, we should also align the objects themselves. In current implementation, this results in rounding the instance size up to 8 bytes. The gap resulting from maintaining this invariant contributes to *external alignment losses*.

Fields layouts are consistent throughout the hierarchy.

Laying out the same fields in the hierarchy at the same offsets enables the VM to skip the field lookups when the superclass field is accessed through different subclasses. This pushes the implementation to lay out the superclass fields first. Even though there could be gaps in superclass field layout that subclasses can subsequently use, current HotSpot implementation does not take this into consideration. The resulting gaps contribute to *internal alignment losses* again.

2. PENDING IMPROVEMENTS

In this study, we try to quantify the scale of alignment losses in realistic classes. We do so to provide the rationale for three pending VM changes:

Close the hierarchy gaps.

There is a pending Request for Improvement (RFE) CR 8024912 [1] which suggests we close the gap between the superclass and subclass instance field blocks. Doing so will eliminate the part of internal alignment losses, and will enable the subclass fields to be put in the alignment *shadow* of the superclass instance block.

Consider this synthetic example:

```
public class A {
  boolean a;
}

public class B extends A {
  boolean b;
}

public class C extends B {
  boolean c;
}
```

The layout for class C will be:

```
OFFSET  SIZE  TYPE DESCRIPTION
     0    12  (object header)
    12     1  boolean A.a
    13     3  (internal align)
    16     1  boolean B.b
    17     3  (internal align)
    20     1  boolean C.c
    21     3  (external align)
```

The only reason we have the internal alignment gaps is because we round up the instance field block in superclass by 4 bytes. We can lay out the fields in this fashion instead:

```
OFFSET  SIZE  TYPE DESCRIPTION
     0    12  (object header)
    12     1  boolean A.a
    13     1  boolean B.b
    14     1  boolean C.c
    15     1  (external align)
```

This will save us 8 bytes per instance. Actually, if we put these three boolean fields in the same class, they will be densely laid out exactly like this.

Re-use the superclass field gaps.

There is also a broader RFE CR 8024913 [2] which suggests we put the subclass fields in every available gap in the superclass, not only on the instance boundary. This requires significant rework of current layout mechanics. Note this RFE is the superset of the RFE above.

Take this example:

```
public class A {
  long a;
}

public class B extends A {
  long b;
}

public class C extends B {
  long c;
  int d;
}
```

The class C will be laid out like this:

```
OFFSET  SIZE  TYPE DESCRIPTION
     0    12  (object header)
    12     4  (internal align)
    16     8  long A.a
    24     8  long B.b
    32     8  long C.c
    40     4  int C.d
    44     4  (external align)
```

Note that we have the gap at offset 12, and we have the *C.a* integer field which can take the place there. We can improve the layout scheme to use that opportunity:

```
OFFSET  SIZE  TYPE DESCRIPTION
     0    12  (object header)
    12     4  int C.d
    16     8  long A.a
    24     8  long B.b
    32     8  long C.c
```

This will save us 8 bytes per instance.

Adjust the alignment.

Another RFE [3] argues that in some cases, we don't need to align the object at 8 bytes, since we don't always have 8-byte fields within the object. This requires the significant rework of the VM-GC interface to communicate the need for different alignment for different objects. Since the prevalent data types are *int* and *reference*, it seems natural keep to 4 bytes as the minimal alignment. Then, if we should differentiate between just two alignments (i.e. 4-byte and 8-byte alignment), we still need to either keep at least a single bit in the object header, or segregate the allocation arenas for the objects with different alignments. Also, in the presence of compressed references, we will have to use the lowest alignment as the compressed reference shift, thus limiting the addressable Java heap.

For example:

```
public class A {
  int a;
}
```

This class does not require the alignment by 8 bytes, the alignment by 4 bytes is enough to guarantee the *int* field alignment:

```
OFFSET  SIZE  TYPE DESCRIPTION
     0     8  (object header)
     8     4  int A.a
```

Rounding up the instance size to 4 bytes, not to 8 bytes gives us 4 spare bytes per instance.

3. EXPERIMENTAL SETUP

- **Java Object Layout (JOL) tools** [6]. In order to quantify the improvements properly, we conduct the study on large corpus of class files, and simulate the layout strategies for the improvements above. The default layout strategies were ported from HotSpot C++ classloader code to Java within the OpenJDK java-object-layout tool, and subsequent changes to the layout policy are also simulated with the help of that tool.

- **Maven Central [7] snapshot**. Taken in early 2012, only JAR artifacts are recorded. 7.9M+ class files, packed in 97K+ JAR files, occupying 66.2 GBs. Since there are multiple versions of the same artifact in the repository, we de-duplicate the extracted layout data, naturally contrasting out only the changed class files.

- **Solaris 10, x86_64**, running on Sun Ultra 27 (1xIntel Xeon W 3540). Since the field layouts are oblivious to OS flavor, we limit our study only to one OS/HW platform.

- **JDK 8b106 Early Access** [8]. We also used the JDK Class Library as one of the auxiliary datasets. The following JVM modes were used:

  - **-d32**: requests 32-bit data model, effectively invoking the 32-bit VM
  - **-d64**: requests 64-bit data model, effectively invoking the the 64-bit VM, compressed references are enabled by default, objects are aligned by 8 bytes.
  - **-d64 -XX:-UseCompressedOops**: same as -d64, but with the compressed references explicitly disabled
  - **-d64 -XX:ObjectAlignmentInBytes=16**: same as -d64, but increasing the object alignment to 16. This is the usual practice to make compressed references work on larger heaps.

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

The largest caveat for this study is the inherent lack of realistic heap dumps. We conduct the study on realistic classes, not the realistic objects, which brings in the sampling bias, since the major part of memory footprint in usual Java application is attributed to just a few classes. However, this study is still important to understand the *worst-case* scenario, when one of those ubiquitous classes is laid out with the losses.

4. BASIC DATA

First, we need to gain the intuition about the usual class sizes. We separate the data for different VMs since they are not usually comparable.

Different JVM modes yield different object header sizes, because we need to get proportionally more metadata in there. The object header in current HotSpot implementation consists of two parts: *mark word*, which aggregates most of the meta-information about the object, and *class word*, which references the class this object is an instance of. Mark word should have the ability to save the large native pointer, i.e. the pointer on stack for the displaced header, or native lock pointer; this blows up the mark word on larger bitnesses. Class word stores the reference to the class, and thus also being affected by bitness.

Header size mandates the minimal instance size for the given JVM mode. The usual sizes are:

- 8 bytes (4 byte mark + 4 byte class) for 32-bit VMs
- 12 bytes (8 byte mark + 4 byte compressed class) for 64-bit VMs with compressed references enabled
- 16 bytes (8 byte mark + 8 byte class) for 64-bit VM without the compressed references

This is coherent with the empirical data on the corpus we have, see Figures 1, 2, 3 and 4. Note that because of the alignment, we have just a few frequent instance sizes, and most of the classes are less than 64 bytes long.

The distribution of *live user data*, that is, space occupied by user fields in the class, is also modalized. This is because the two most prevalent data types used, *int*, and *reference*. Notably, the only JVM mode which has different user data footprint is the one with compressed references disabled. In all other cases, the sizes of all built-in types are exactly the same, and so the user data takes the same space. See Appendix A for more details on user data.

By comparing instance sizes vs. live user data we can infer the alignment losses. In many cases the alignment losses are exactly zero. However, because user data is not fitting exactly in the aligned object, we need to waste some of the space.

The alignment losses generally depend on object alignment and header size. For example, in 32-bit VM (Figure 5), we can see most of the objects enjoy no alignment losses at all, while we still have the loss peak at 4 bytes. This is usual when using single *reference* field with 8 byte header, requiring the object to be aligned up by 8 bytes. In 64-bit VM case we have the 16-byte header, and 8-byte references. While the objects are generally larger, having the full-width references saves us from a large quantity of alignment losses (see Figure 7). This effect is back with compressed references enabled (see Figure 6). Naturally, this effect is amplified even more with 16-byte alignments (see Figure 8).

```
BitSet claimed = new BitSet();
claimed.set(0, OBJ_HEADER_SIZE);

for (Class<?> k : hierarchy) {
  for (int size : new int[] {8, 4, 2, 1}) {
    for (FieldInfo f : enumerateFields(k)) {
      if (f.getSize() != size) continue;

      // Find the next free slot, and
      // make sure it is $size-aligned.
      for (int t = 0; t < Integer.MAX_VALUE; t++) {
        int start = t*size;
        int end   = (t+1)*size;
        if (claimed.get(start, end).isEmpty()) {
          claimed.set(start, end);
          f.setOffset(start);
          break;
        }
      }
    }
  }

  if (doSuperGaps) {
    // Do nothing, allowing to use any gaps
    // in the superclass layout.
  } else if (doHierarchyGaps) {
    // Claim the entire class, no rounding.
    // This will allow using the hierarchy
    // alignment shadow.
    claimed.set(0, claimed.lastBitSet());
  } else {
    // Claim the entire class,
    // also round up the instance field block.
    // This is default for HotSpot today.
    claimed.set(0, align(claimed.lastBitSet(), 4));
  }
}

instanceSize = claimed.lastBitSet() + 1;
if (autoAlign) {
  int a = 4;
  for (FieldInfo f : enumerateFields(k)) {
     a = Math.max(a, f.getSize());
  }
  instanceSize = align(instanceSize, a);
} else {
  instanceSize = align(instanceSize, 8);
}
```

Figure 9: Simulated field layout code

We can also note the non-power-of-two alignment losses due to the non-uniform user data. These losses can occur on the hierarchy boundary, or because the first field alignment. In this study, we will estimate the benefits of avoiding these alignment losses.

5. LAYOUT SCHEMES

Current HotSpot classloader code [9] is responsible for laying out the fields in the class. HotSpot clearly demarcates instance and static fields: while instance fields are the part of the object instance, the static fields are the part of class data. The current code is complicated due to different allocation styles, the presence of hard-coded field offsets for some classes, @Contended, etc.

In this experiment, we reformulated the layout code into the Java code in Figure 9. Note this code is laying out the entire hierarchy at once while still maintaining the impor-

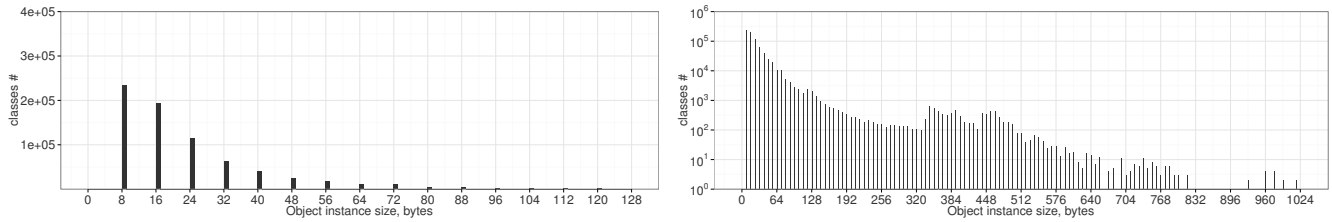

Figure 1: Instance size distribution on 32-bit VM: 8-byte headers, objects aligned by 8 bytes.

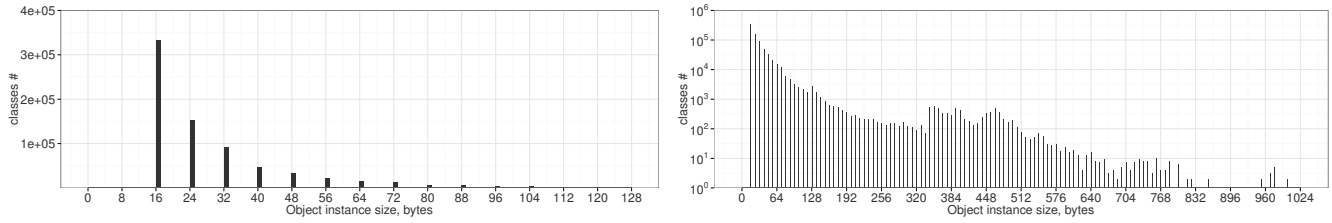

Figure 2: Instance size distribution on 64-bit VM: 12-byte headers, objects aligned by 8 bytes.

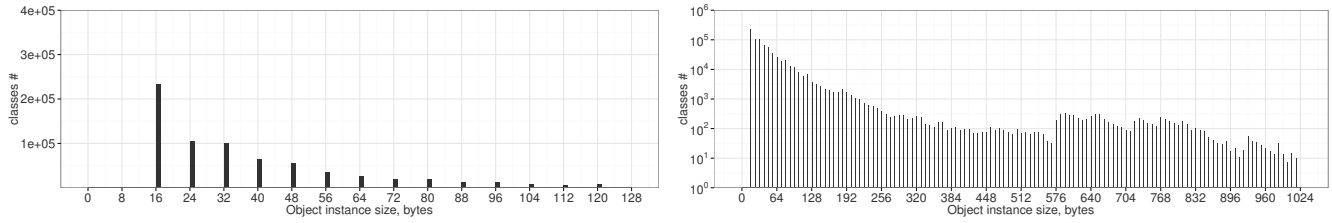

Figure 3: Instance size distribution on 64-bit VM, compressed refs disabled: 16-byte headers, objects aligned by 8 bytes.

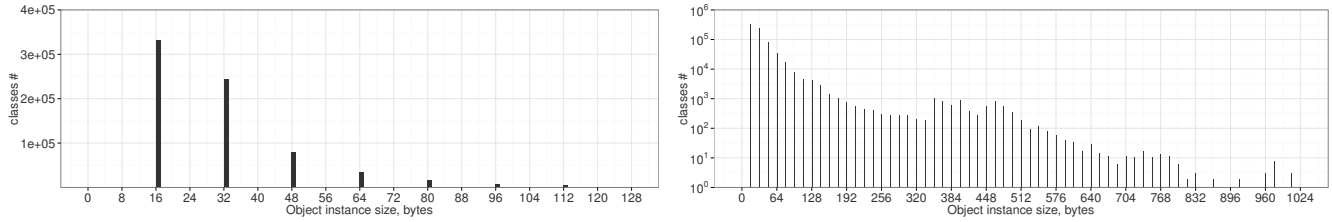

Figure 4: Instance size distribution on 64-bit VM, object alignment 16: 16-byte headers, objects aligned by 16-bytes.

tant invariants (see Section 1), and this approach is different from HotSpot's one class at the time, starting from superclass. Our code maintains the bit set of claimed offsets in the instance field block, traverses the hierarchy from superclasses down to subclasses, and lays out the instance fields from widest to narrowest by finding the appropriate slot.

Note that our code is rather ineffective since it takes $O(n^2)$ time, and $O(n)$ space, where n is the number of instance fields. Current HotSpot layout code has $O(n)$ time, and $O(1)$ space complexity. A few optimizations can provide $O(n)$ time and space complexity for our naive algorithm.

We have checked that the instance sizes generated by our simplistic layout are coherent with the real VM layout: less than 0.5% classes produce different layout on the entire corpus. These discrepancies are caused by Throwable.backtrace field not visible via Java Reflection [4], and also because of @Contended [5]. See Appendix B for more data.

We suggest three major improvements to this layout scheme.

First, we can stop rounding up the current class instance block to 4, thus making the hierarchy gap accessible for the layout. We refer to this strategy as "hierarchy gap". This strategy it is enabled by flipping the "doHierarchyGap" flag in Figure 9.

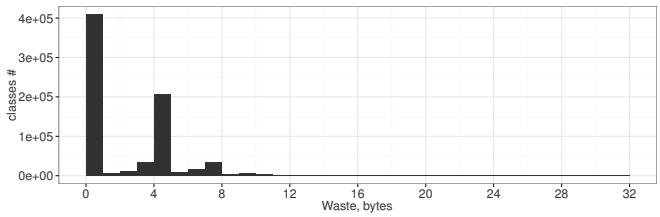

Figure 5: Alignment losses on 32-bit VM

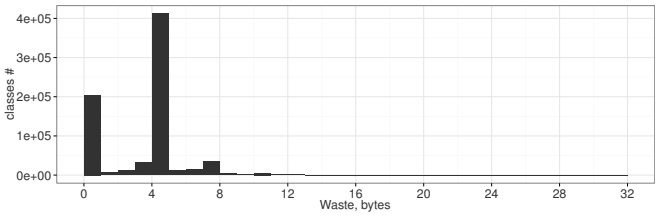

Figure 6: Alignment losses on 64-bit VM, compressed references enabled

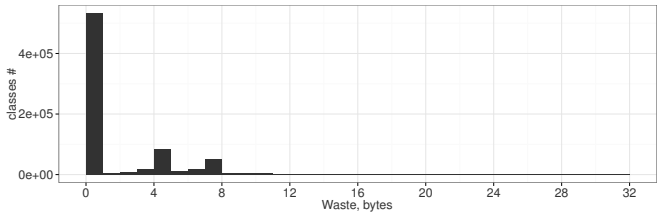

Figure 7: Alignment losses on 64-bit VM, compressed references disabled

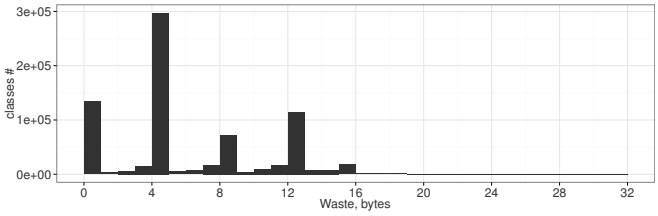

Figure 8: Alignment losses on 64-bit VM, compressed references enabled, object alignment is 16

Second, we can stop claiming the entire class after the layout, making its gaps available for subclasses. We refer to this strategy as "super gap". This strategy is enabled by flipping the "doSuperGap" flag in Figure 9. "Super gap" strategy supersedes the "hierarchy gap" strategy, because the former naturally claims the hierarchy gaps as well.

Third, we can figure out the maximum field width in the current class, and infer the required alignment for this object. This feature is enabled by flipping the "autoAlign" flag in Figure 9. This is a generic optimization applicable to both "hierarchy"- and "super gap" layout strategies.

6. RESULTS

By applying different layout strategies to the existing classes, we can showcase the benefits of introducing these changes into the JVM. Since the alignment losses are different in distinct JVM modes, we can foresee the expected improvements are also different.

The experimental data are summarized in Figure 10. Up to 1.8% of real classes benefit from even the simple "hierarchy gap" strategy. Up to 2.4% of all encountered classes benefit from the more aggressive "super gap" strategy. Both these optimizations shift the instance size to the lower alignment. Because of that, the improvements are multiples of 8 bytes, with the most frequent improvement (>99% of classes) being 8 bytes, see Figure 11. Due to the fact the most prevalent classes are small, the relative improvements are moderate, around 10% in default configurations, see Figure 12.

The most interesting result, however, comes from the automatic alignment. We can see that even the auto alignment alone can be applied for more than 30% of realistic classes, resulting in dramatic footprint improvements. The major improvements in most VM modes are because of the fact that user data is not normally occupying the entire alignment shadow. 64-bit mode with compressed references disabled experiences much less improvements, because the references are now aligned by 8 bytes, and have better chances to fit the alignment. The average improvements for this op-

	32-bit VM	64-bit VM		
		+comp	−comp	16-byte
hierarchy	1.6%	1.5%	1.8%	0.6%
super	1.8%	1.8%	2.4%	0.7%
align	32.7%	60.4%	1.3%	76.1%
align + hierarchy	32.3%	62.0%	3.1%	77.0%
align + super	34.4%	62.2%	3.7%	77.1%

Figure 10: Maven Central: Optimizable classes ratio

	32-bit VM	64-bit VM		
		+comp	−comp	16-byte
hierarchy	8.0	8.0	8.0	16.0
super	8.1	8.0	8.0	16.0
align	4.0	4.0	4.0	6.7
align + hierarchy	4.2	4.1	6.3	6.8
align + super	4.2	4.1	6.6	6.8

Figure 11: Maven Central: Average instance size improvement, bytes

	32-bit VM	64-bit VM		
		+comp	−comp	16-byte
hierarchy	9.7%	9.4%	6.0%	15.0%
super	9.8%	9.8%	6.4%	15.9%
align	17.6%	19.1%	15.8%	23.4%
align + hierarchy	17.2%	18.8%	10.2%	23.3%
align + super	17.2%	18.8%	9.7%	23.3%

Figure 12: Maven Central: Average instance size improvement, percent

	32-bit VM	64-bit VM		
		+comp	−comp	16-byte
hierarchy	1.5%	1.3%	1.0%	0.6%
super	1.7%	1.6%	1.7%	0.7%
align	34.4%	58.1%	2.4%	75.1%
align + hierarchy	35.9%	59.4%	3.4%	75.8%
align + super	36.0%	59.6%	4.1%	75.9%

Figure 13: JDK 8: Optimizable classes ratio

	32-bit VM	64-bit VM		
		+comp	−comp	16-byte
hierarchy	8.0	8.0	8.0	16.0
super	8.0	8.0	8.1	16.0
align	4.0	4.0	4.0	6.8
align + hierarchy	4.2	4.1	5.2	6.8
align + super	4.2	4.1	5.1	6.9

Figure 14: JDK 8: Average instance size improvement, bytes

	32-bit VM	64-bit VM		
		+comp	−comp	16-byte
hierarchy	10.6%	9.8%	5.3%	15.5%
super	10.5%	10.0%	5.8%	15.2%
align	16.9%	19.1%	16.0%	23.0%
align + hierarchy	16.7%	18.9%	12.8%	23.0%
align + super	16.7%	18.7%	11.7%	23.0%

Figure 15: JDK: Average instance size improvement, percent

timization are ranging in 10% – 20% reduction in instance size.

Figures 13, 14, and 15 show the related improvements for the JDK class library (conveniently packed into rt.jar) shipped with the JDK. We can see that the improvements are slightly lower than those measured with Maven Central, but otherwise coherent with them.

7. CONCLUSION AND DISCUSSION

This report quantified the world-wide impact for three possible VM improvements.

We have shown that taking the hierarchy gaps can help more than 1.8% of real-world classes to occupy 8 bytes less per instance. We had also shown that more generic optimization, i.e. taking the superclass gaps can help more than 2.4% of real-world classes to occupy 8 byte less per instance. These two features only touch the specific part in JVM runtime, and do not require heavy runtime support.

Additionally, we demonstrated that current object alignment of 8 bytes is very pessimistic, and more than 30% classes in 32-bit mode, or 60% classes in 64-bit mode can benefit from the lower alignment, saving 4 bytes and more per instance. This optimization requires major changes in runtime to support multiple alignments, including, but not limited to, storing the alignment flags in object headers, adjusting GC to take care of multiple alignments, as well as solving the potential problems with misaligned object header accesses.

The improvements are measured on the large corpus of realistic class files, not the large corpus of realistic objects. The actual footprint improvements will depend on the object distribution in particular applications, and requires further research. There is the observation that just a few classes occupy most of the heap in realistic applications. The data in this study gives the rough estimate for the probability we hit the unlucky layout with some of those frequent classes, and the penalties in the memory footprint we will encounter.

8. ACKNOWLEDGMENTS

We would like to thank Mandy Chung, who had the Maven Central dump handy for quick analysis. We also thank Sergey Kuksenko, Dmitry Chuyko, Staffan Friberg, and Claes Redestad for the draft reviews, comments and suggestions.

9. REFERENCES

[1] *CR 8024912: Subclass instance fields are laid out with alignment gaps*, `http://bugs.openjdk.java.net/browse/JDK-8024912`

[2] *CR 8024913: Subclasses are not using the super-class space gaps to place the fields*, `http://bugs.openjdk.java.net/browse/JDK-8024913`

[3] *CR 8025677: Multiple alignments for Java objects*, `http://bugs.openjdk.java.net/browse/JDK-8025677`

[4] *CR 4496456: Segmentation fault introspecting ((Throwable) obj).backtrace[0][0]*, `http://bugs.openjdk.java.net/browse/JDK-4496456`

[5] *JEP 142: Reduce Cache Contention on Specified Fields*, `http://openjdk.java.net/jeps/142`

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

[7] *Maven Central repository*, `http://repo1.maven.org/`

[8] *JDK 8 Early Access Releases* `https://jdk8.java.net/download.html`

[9] *HotSpot classloader code, see "layout_fields" method* `http://hg.openjdk.java.net/jdk8/jdk8/hotspot/file/tip/src/share/vm/classfile/classFileParser.cpp`

APPENDIX

A. LIVE USER DATA SIZES

Figures 16 and 17 show the usual live user data sizes for the classes in our corpus.

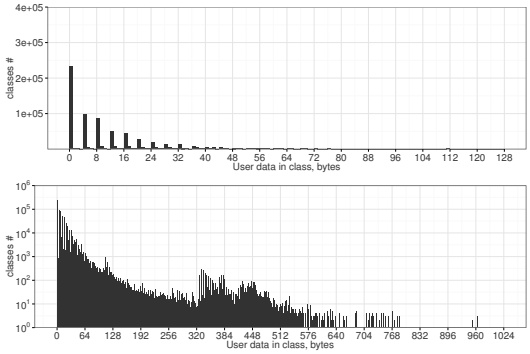

Figure 16: User data with 4-byte references.

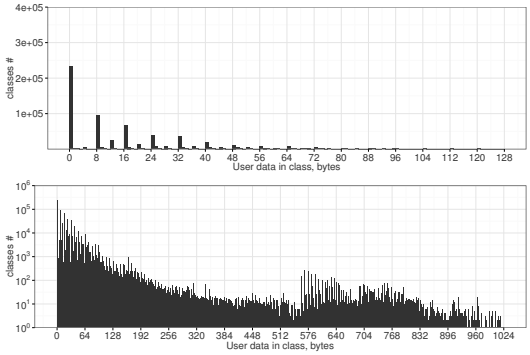

Figure 17: User data with 8-byte references.

B. VM LAYOUT DISCREPANCIES

Figures 18, 19, 20, and 21 show the discrepancies between the the real VM layout, and our simulation code. The peaks at 128 and 256 bytes are @Contended [5] peaks. The peak at 8 bytes is the Throwable.backtrace [4] peak. There are also a few other peaks at Figure 19 attributed to hard-coded field offsets in *java.lang.Class* and *java.lang.ClassLoader*.

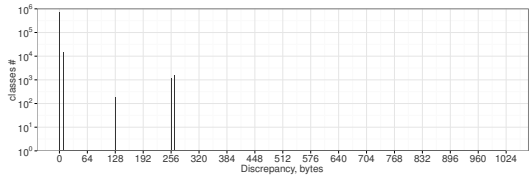

Figure 18: VM layout discrepancies for 32-bit VM

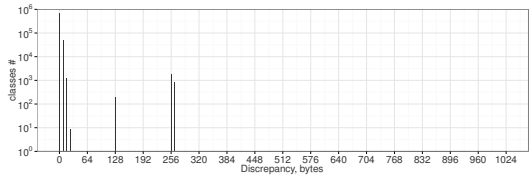

Figure 19: VM layout discrepancies for 64-bit VM without the compressed references

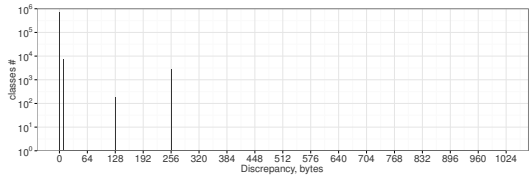

Figure 20: VM layout discrepancies for 64-bit VM with the compressed references

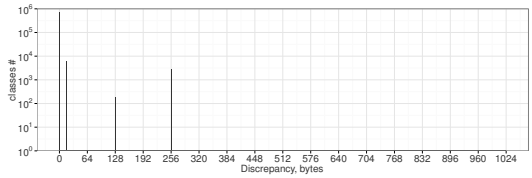

Figure 21: VM layout discrepancies for 64-bit VM with 16-byte alignment