Main

Project Lilliput

Goals

1. Reduce the object header to 64 bits. It may be possible to shrink it down to 32 bits as a secondary goal.
2. Make the header layout more flexible, i.e. allow some build-time (possibly even run-time) configuration of how we use the bits.

Motivation

In 64-bit Hotspot, Java objects have an object header of 128 bits: a 64 bit multi-purpose header (â€˜markâ€™ or â€˜lockâ€™) word and a 64-bit class pointer. With typical average object sizes of 5-6 words, this is quite significant: 2 of those words are always taken by the header. If it were possible to reduce the size of the header, we could significantly reduce memory pressure, which directly translates to one or more of (depending what you care about or what your workload does):

- Reduced heap usage
- Higher object allocation rate
- Reduced GC activity
- Tighter packing of objects -> better cache locality

In other words, we could reduce the overall CPU and/or memory usage of all Java workloads, whether it's a large in-memory database or a small containerized application.

Current situation

The object header (in 64 bit Hotspot builds) is currently 128 bits long. The first 64 bits are the so-called 'lock' or 'mark' word, the subsequent 64 bits are the class-pointer.

```
+------------------+
| Lock-word        |
+------------------+
| Class pointer    |
+------------------+
| Field 1          |
+--------+---------+
| Field 2| Field 3 |
+--------+---------+
| etc              |
```

For arrays, we will reserve an additional field for the arraylength:

```
+------------------+
| Lock-word        |
+------------------+
| Class pointer    |
+------------------+
| Array-Length     |
+--------+---------+
| Elem 1 | Elem 2  |
+--------+---------+
| etc              |
```

The lock/mark word

The lock word (for simplicity I'll continue to call it that, even if it's grossly imprecise) is overloaded with various purpuses:

- Locking: The 3 lowest bits are used for locking, and can take the following combinations:
    - `[ ptr        | 00 ]` Locked, the upper bits interpreted as a pointer point to real header on stack
    - `[ header | 0 | 01 ]` Unlocked, upper bits are regular object header
    - `[ ptr        | 10 ]` Monitor, the upper bits point to inflated lock, header is swapped out
    - `[ 0 ..... 0 | 00 ]` Inflating in progress
    - `[ ptr        | 11 ]` Forwarded, used by GC to indicate that upper bits point to forwarded object, which also contains the real header
- The 3rd lowest bit and the following two states are used for biased locking, which is deprecated and will eventually be removed:
    - `[ JavaThread* | epoch | age | 1 | 00 ]` Biased towards given thread
    - `[ 0           | epoch | age | 1 | 00 ]` Anonymously biased
- GC:
    - `Generational GCs use bits 4-7 for tracking object age:`

      `[ ... | age 4 bits | 0 | 01 ]`

    - Some GCs use the header to point to the relocated object during relocation:

      `[ ptr | 11 ]` Forwarded, used by GC to indicate that upper bits point to forwarded object, which also contains the real header
- Identity hashcode: First call to System.idendityHashCode() computes the i-hash and stores it into the upper bits of the header

  `[ 25 bits unused | 31 bits i-hash | age 4 bits | 0 | 01 ]`

The class-pointer

The class-pointer can either be a regular pointer, pointing to the corresponding metaspace Klass instance, or it can be a 32 bit compressed class pointer, which, when uncompressed points to the corresponding Klass instance. In the latter case, we have 32 unused bits, which may be taken by arrays to store their arraylength, which is also 32 bits.

Constraints

- Performance
- If we limit e.g. number of classes/monitors/etc that we can encode, we need a way to deal with overflow
- Requires changes in assembly across all supported platforms (also consider 32 bits)
- Interaction with other projects like Panama, Loom, maybe Leyden, etc
- System.identityHashcode() is specified as 32bit integer. We may want to use more bits, e.g. 64 bit or even 128 bits to improve hash distribution, but that would require very significant spec changes and would affect Object.hashCode() and all sorts of Hash* java.util collections.
- array-length is specified as 31 bit integer. We would like to be able to address larger arrays, but that is a difficult spec change.

Possible approaches

We have a wide variety of techniques to explore for allocating and down-sizing header fields

- Pointers can be compressed, e.g. if we expect a maximum of, say, 8192 classes, we could, with some careful alignment of Klass objects, compress the class pointer down to 13 bits: 2^13=8192 addressable Klasses. Similar considerations apply to stack pointers and monitors.
- Instead of using pointers, we could use class IDs that index a lookup table
- We could backfill fields which are known at compile-time (e.g. alignment gap or hidden fields)
- We could use backfill fields appended to an object after the GC moved it (e.g. for hashcode)
- We could use side-tables

Class-pointer

Class-pointers are accessed for all type checks and are also used to resolve virtual calls at runtime. They are probably the most performance-sensitive part of the header. There are several approaches to downsizing the class-pointer field:

- Compressed class-pointer: current class-pointer compression uses 32bit and can address up to 32GB of address-space which is much more than we would ever need. Realistically, we should be able to compress the pointer to 20 bits, which would give us 2^20=1GB addressable space, assuming we can smartly align Klass instances and even-layout them. More tight compressions also requires to split the Klass-structure into a fixed-size near-class and a variable-sized far-class. The near-class would then be addressed by the compressed pointer, and the far-class be referenced by the near-class. Unfortunately, part of the variable-sized part involves the vtable, which may be too performance-sensitive.
- Class-index: header carries index into class-table which in turn points to the class objects. This means an additional indirection for all accesses of the class.

Locking

It is interesting to note that only relatively few (<1%) Java objects are ever used for locking. It seems useful to not let all Java objects carry unnecessary weight for locking. Also, biased-locking is deprecated and will eventually be removed.

- With an eye to project Loom it seems useful to consider using Java locks and put away with most of runtime locking machinery. This requires associating a ReentrantLock-(Java-) object with the locked object, and call into its corresponding locking methods. It also requires to make compiled code to efficiently find the lock object. It also requires that GC knows about the object->lock-object association.
- Ongoing work on compact java locks
- We could perhaps continue with current locking scheme, however any header displacement means that class-lookup code needs to be able to efficiently find the class-pointer, which may be problematic performance wise.

Hashcode

It is interesting to note that with most workloads, only relatively few (<1%) Java objects are ever assigned an i-hash. It seems useful to not let all Java objects carry unnecessary weight for hashing.

- We could use fewer bits for hashcode. However, we'd probably rather use more (64 bits, or even 128 bits) to dodge the birthday paradoxon.
- We could recompute the i-hash as long as an object doesn't move. As soon as the GC moves it, we could indicate that fact in a dedicated bit, and append an i-hash word somewhere (to be determined by the type of object, could be an alignment gap, or an appendix to the original object). This is somewhat complex, but avoids allocating an extra word for i-hash for many many objects.
- We could store the i-hash in a side-table of sorts.
- We can also reduce the size of the header for certain kind of classes, by example for a record, we know that the field are truly final so we can avoid to compute the hashCode and use the fields to calculate the identity hashCode the same way Valhalla does for the primitive classes.

- For a primitive class, when they are on the heap, again, we can avoid the identity hashCode (and also the lock bits, but that's less interresting).

GC

- We will probably keep using the age bits
- We can probably keep using the forwarding mechanism:
    - Shenandoah can store the forwarding pointer safely in the old object copy - the real header is in the new copy anyways.
    - ZGC already uses side-tables for forwarding information and doesn't use the header at all.
    - Other GCs will swap out the header during compaction, and copy it back to the object copy.
    - It requires 64 bits though, if we can do even smaller headers, we need to think about side-tables.

Proposed 'real-quick' (haha) prototype

- Use ZGC (requires no header bits), but assume we're going to require age bits and forwarding pointer
- Smaller hashcode
- 32 bit compressed class-pointer
- Keep current locking scheme. Requires work to get the actual class-ptr in the face of displaced headers. Otherwise can use a lot of existing machinery.

Suggestions welcome! :-)

Interferences

Valhalla

Valhalla has the potential to reduce the impact of Lilliput, it reduces memory usage by flattening object graphs into a packed layout. However, Lilliput is orthogonal and complementary to Valhalla. Valhalla may need a bit or two in the header too.

Loom

Loom will very likely affect how we want to do locking. We should consider this when choosing different approaches.

Research

Some workloads have been run with an instrumented JVM that gives us some information about object sizes, potential savings, number of hashes and locks, etc. See this table for details.

Work plan

- Research (partly done, see above): First we should conduct some experiments to establish some factors that will likely affect our implementation choices. These measurements can be conducted by instrumenting the JVM to collect various statistics.
    - Object-size histogram: How much space would we realistically save?
    - How many objects are typically i-hashed?
    - How many objects are typically locked?
    - How many classes are typically loaded? (avg/max)
    - How performance-sensitive is access to the Klass of an object?
- These experiments should be conducted across as many different workloads as we can get hold of. A good start would be a set of benchmarks that typically test a good number of different workloads:
    - SPECjvm2008
    - Dacapo
    - Renaissance
    - SPECjbb2015
- Other workloads that come to mind include:
    - Various IDEs: Eclipse, Netbeans, IntelliJ
    - Minecraft
    - Application servers, web servers
    - Spring Petclinic
    - TechEmpower Web Framework Benchmarks
- Based on the above research, decide how to approach the first goal of 64 bit headers:
    - Can we use compressed class-pointers, possibly with fewer bits than 32, or do we need (or can we even afford?) an extra indirection via Klass lookup-table?
    - How do we approach locking? Can we compress pointers to stack locks, or can we avoid stack-locks altogether? Same for inflated locks?
    - How do we approach i-hashing?
    - (How?) Can we implement dynamic allocation of extra fields, e.g. for i-hash or maybe even for locking-support?
    - Where does arraylength fit?
- Implement improved i-hashing
- Implement improved locking
- Implement improved Klass*
- Wire it all together and collapse header to 64 bits

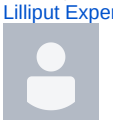
- Future work: 32 bit or even smaller header? (If so: improve field layout to put fields in unused bits of header word)

Resources

- Lilliput Project
- Repository
- Mailing list: lilliput-dev (archives)
- Lilliput CFD
- JEP
- Lilliput Experiment Results
- Gil Tene about compressing class-pointers
- Type Information Elimination from Objects on Architectures with Tagged Pointers Support

Recent space activity

Roman Kennke
Main updated Apr 12, 2023 view change
Lilliput Experiment Results created Apr 12, 2023



Iris Clark
Main updated Apr 08, 2021 view change

Space contributors

- Roman Kennke (58 days ago)
- Iris Clark (793 days ago)