JEP 387: Elastic Metaspace

Owner	Thomas Stuefe
Type	Feature
Scope	JDK
Status	Closed / Delivered
Release	16
Component	hotspot / runtime
Discussion	hotspot dash runtime dash dev at openjdk dot java dot net
Reviewed by	Coleen Phillimore, Goetz Lindenmaier
Endorsed by	Mikael Vidstedt
Created	2019/03/20 18:22
Updated	2023/08/15 15:35
Issue	8221173

Summary

Return unused HotSpot class-metadata (i.e., *metaspace*) memory to the operating system more promptly, reduce metaspace footprint, and simplify the metaspace code in order to reduce maintenance costs.

Non-Goals

- It is not a goal to change the way that compressed class-pointer encoding works, or the fact that a compressed class space exists.
- It is not a goal to extend the use of the metaspace allocator to other areas of HotSpot, though that may be a possible future enhancement.

Motivation

Since its inception in JEP 122, metaspace has been somewhat notorious for high off-heap memory usage. Most normal applications don't have problems, but it is easy to tickle the metaspace allocator in just the wrong way to cause excessive memory waste. Unfortunately these types of pathological cases are not uncommon.

Metaspace memory is managed in per-class-loader arenas. An arena contains one or more *chunks*, from which its loader allocates via inexpensive pointer bumps. Metaspace chunks are coarse-grained, in order to keep allocation operations efficient. This can, however, cause applications that use many small class loaders to suffer unreasonably high metaspace usage.

When a class loader is reclaimed, the chunks in its metaspace arena are placed on freelists for later reuse. That reuse may not happen for a long time, however, or it may never happen. Applications with heavy class loading and unloading activity can thus accrue a lot of unused space in the metaspace freelists. That space can be returned to the operating system to be used for other purposes if it is not fragmented, but that's often not the case.

Description

We propose to replace the existing metaspace memory allocator with a buddy-based allocation scheme. This is an old and proven algorithm which has been used successfully in, e.g., the Linux kernel. This scheme will make it practical to allocate metaspace memory in smaller chunks, which will reduce class-loader overhead. It will also reduce fragmentation, which will allow us to improve elasticity by returning unused metaspace memory to the operating system.

We will also commit memory from the operating system to arenas lazily, on demand. This will reduce footprint for loaders that start out with large arenas but do not use them immediately or might never use them to their full extent, e.g., the boot class loader.

Finally, to fully exploit the elasticity offered by buddy allocation we will arrange metaspace memory into uniformly-sized *granules* which can be committed and uncommitted independently of each other. The size of these granules can be controlled by a new command-line option, which provides a simple way to control virtual-memory fragmentation.

A document describing the new algorithm in detail can be found here. A working prototype exists as a branch in the JDK sandbox repository.

Alternatives

Instead of modernizing metaspace, we could remove it and allocate class metadata directly from the C heap. The advantage of such a change would be reduced code complexity. Using the C-heap allocator would, however, have the following disadvantages:

- As an arena-based allocator, metaspace exploits the fact that class metadata objects are bulk-freed. The C-heap allocator does not have that luxury, so we would have to track and release each object individually. That would increase runtime overhead, and, depending on how the objects are tracked, code complexity and/or memory usage.
- Metaspace uses pointer-bump allocation, which achieves very tight memory packing. A C-heap allocator typically incurs more overhead per allocation.
- If we use the C-heap allocator then we could not implement the compressed class space as we do today, and would have to come up with a different solution for compressed class pointers.
- Relying too much upon the C allocator brings its own risk. C-heap allocators can come with their own set of problems, e.g., high fragmentation and poor elasticity. Since these issues are not under our control, solving them requires cooperation with operating-system vendors, which can be time-intensive and easily negate the advantage of reduced code complexity.

Nevertheless, we tested a prototype that rewired metadata allocation to the C heap. We compared this `malloc`-based prototype to the buddy-based prototype, described above, running a micro-benchmark which involved heavy class loading and unloading. We switched off the compressed class space for this test since it would not work with C-heap allocation.

On a Debian system with glibc 2.23, we observed the following issues with the `malloc`-based prototype:

- Performance was reduced by 8-12%, depending on the number and size of loaded classes.
- Memory usage (process RSS) increased by 15-18% for class load peaks before class unloading.
- Memory usage did not recover at all from usage spikes, i.e., metaspace was completely inelastic. This led to a difference in memory usage of up to 153%.

These observations hide the memory penalty caused by switching off the compressed class space; taking that into consideration would make the comparison even more unfavorable for the `malloc`-based variant.

Risks and Assumptions

Virtual-memory fragmentation

Every operating system manages its virtual memory ranges in some way; the Linux kernel, e.g., uses a red-black tree. Uncommitting memory may fragment these ranges and increase their number. This may affect the performance of certain memory operations. Depending on the OS, it also may cause the VM process to encounter system limits on the maximum number of memory mappings.

In practice the defragmentation capabilities of the buddy allocator are quite good, so we have observed a very modest increase in the number of memory mappings. Should the increased number of mappings be a problem then we would increase the granule size, which would lead to coarser uncommitting. That would reduce the number of virtual-memory mappings at the expense of some lost uncommit opportunities.

Uncommit speed

Uncommitting large ranges of memory can be slow, depending on how the OS implements page tables and how densely the range had been populated before. Metaspace reclamation can happen during a garbage-collection pause, so this could be a problem.

We haven't observed this problem so far, but if uncommit times become an issue then we could offload the uncommitting work to a separate thread so that it could be done independently of GC pauses.

Reclamation policy

To deal with potential problems involving virtual memory fragmentation or uncommit speed, we will add a new production command-line option to control metaspace reclamation behavior:

```
`-XX:MetaspaceReclaimPolicy=(balanced|aggressive|none)`
```

- `balanced`: Most applications should see an improvement in metaspace memory footprint while the negative effects of memory reclamation should be marginal. This mode is the default, and aims for backward compatibility.
- 'aggressive': Offers increased memory-reclamation rates at the cost of increased virtual-memory fragmentation.
- 'none': Disables memory reclamation altogether.

Maximum size of metadata

A single metaspace object cannot be larger than the *root chunk size*, which is the largest chunk size that the buddy allocator manages. The root chunk size is currently set to 4MB, which is comfortably larger than anything we would want to allocate in metaspace.