Metaspace in OpenJDK 16

Leo Korinth

Thomas Stüfe (working for SAP) has contributed a new Metaspace as described in JEP 387. This is a great example of an improvement made to OpenJDK by contributors outside of Oracle.

Metaspace improvements in OpenJDK 16

When you start using Java 16, you will benefit from the new Metaspace — without doing anything.

- Fragmentation in Metaspace will be reduced, resulting in lower memory usage.
- When memory is not used any more, there is a much higher probability that it can be returned to the operating system, which will reduce the memory footprint of the JVM process.

If you are doing lots of class loading and want to trim memory usage, this change should be of interest to you! Before this change, small class loaders wasted lots of memory due to inflexibility in chunk sizes; for small class loaders *huge* savings can be made in Metaspace. Even in normal applications such as WildFly or Eclipse, memory reductions can be significant (>10%) in Metaspace. Take a look at the presentation by Thomas Stüfe, statistics starting at page 47.

Metaspace background

Metaspace is the name of a memory allocator (or the space used by the allocator). It allocates memory outside the Java heap. This memory is mostly used for class metadata, but it could be used as a general arena allocator as well. Prior to JDK 8, metadata was allocated in a special part of the Java heap called the permanent generation, or PermGen.

An allocator needs to be able to allocate and — most often — free memory. Metaspace has one additional important feature in that you can make it allocate memory within a specified *address space*. With this feature, each Java object can reach class data using a 32-bit index instead of wasting a 64-bit pointer on *each object header*. This feature is called compressed class pointers and works very similar to compressed oops.

Class loading/unloading mostly follow the allocation pattern of allocating data without freeing until the classloader is discarded (and then everything is freed). This usage pattern allows us to tune the allocator somewhat in addition to support compressed class pointers.

Why not use an ordinary allocator such as malloc?

The generic malloc does not know how to allocate in a restricted memory space (for compressed class pointers). So either all references to class metadata must go through an indirection table (requires compiler changes, with a possible performance impact) or we need a special malloc. Metaspace can be seen as this special malloc that knows how to allocate to a small address space. Using malloc was tested and rejected.

Short walk-through

Chunk allocation using a buddy allocator

Memory is allocated from the operating system in `VirtualSpaceNode`s. When not allocating into compressed class space (blue), these will be stored in one linked list with many nodes. When allocating into compressed class space (orange), the list will contain precisely one `VirtualSpaceNode` reserving a 32-bit address space thus guaranteeing access with compressed pointers. Although the whole address space is reserved at once, memory is committed lazily only when being used.

Each `VirtualSpaceNode` will have a bitmap keeping track of which commit granules have been committed, more information on this further down.

Every `VirtualSpaceNode` is divided into several, in address space consecutive `RootChunkArea`s, currently sized 4 MiB. This is the max size of a memory allocation. A `RootChunkArea` is an area that can be recursively halved into smaller areas of logarithmic size, 4, 2, 1, ½, ... MiB. These chunks can cheaply be removed or merged back using a buddy allocator scheme. When added back, the chunk will try to recursively merge into its neighbour *buddy* reducing *external* fragmentation.

In the general case, the JVM will have two instances of `VirtualSpaceNode` lists, one with a compressed space where class data is put, and one uncompressed that is not limited by address space and used for everything not needing compressed class pointers.

Class loader allocation and destruction

The JVM always allocates through a class loader. A JVM contains many class loaders; they are created and destroyed during the lifetime of the JVM process. Each class loader will allocate chunks from the two JVM-global

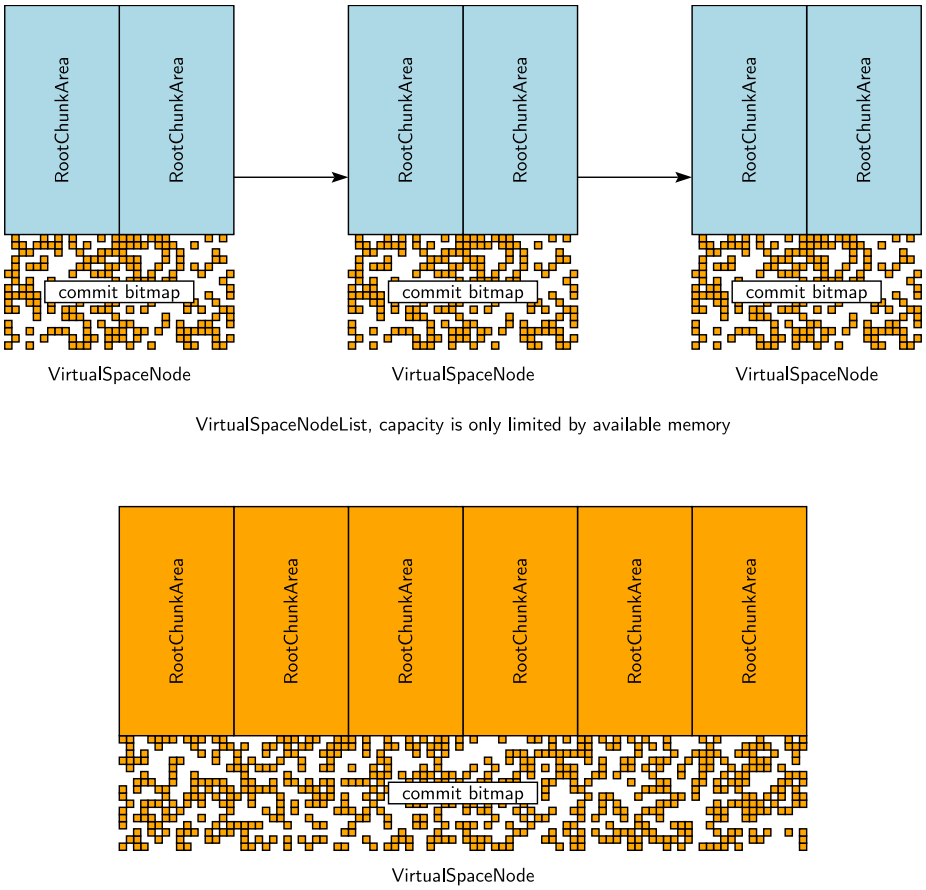


Figure 1: A JVM running with compressed class pointers will have two `VirtualSpaceNodeLists`, one for "normal" objects, one for objects that needs to be addressed by compressed class pointers.

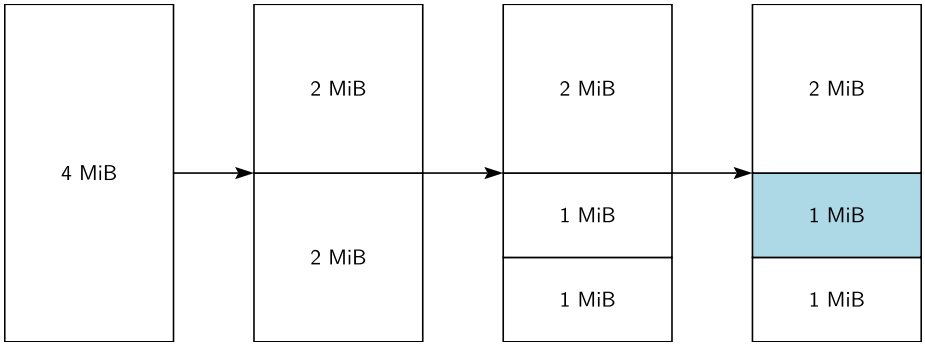


Figure 2: Blue class loader allocates a 1 MiB block, the root chunk needs to be divided twice.

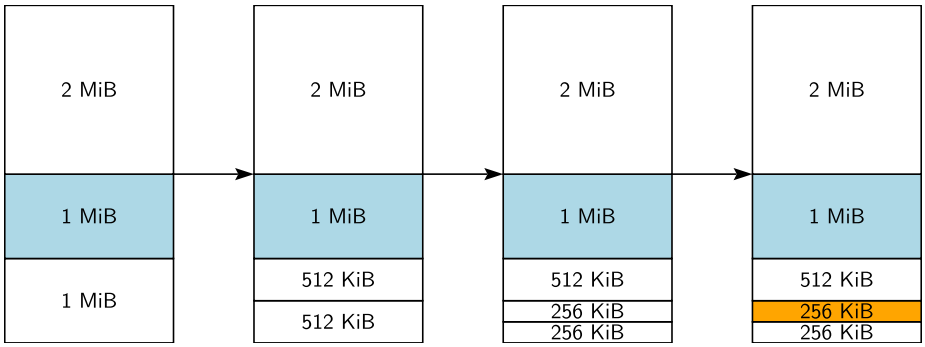


Figure 3: Orange classloader allocates 256 KiB; note that both class loaders might allocate chunks from the same `RootChunkArea`.

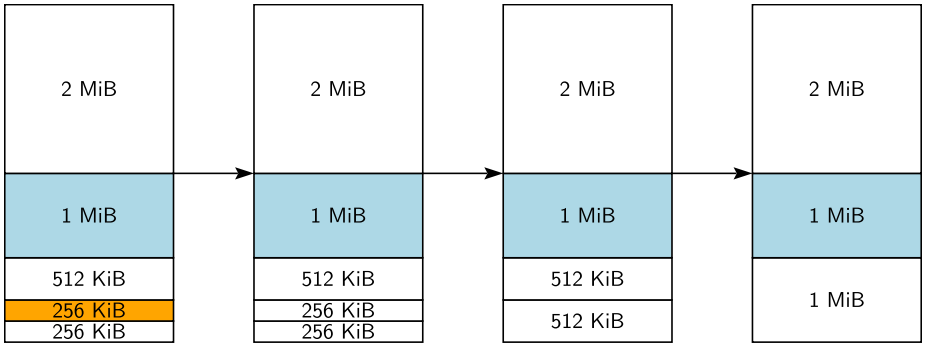


Figure 4: When the orange class loader is destroyed, the free 256 KiB chunk will merge with its buddy chunk twice, leaving big consecutive areas without external fragmentation.

`VirtualSpaceLists`. From these big chunks of memory, the amount of bytes needed for the current allocation will be bump allocated to the user. The remainder of the chunk can be bump allocated from later.

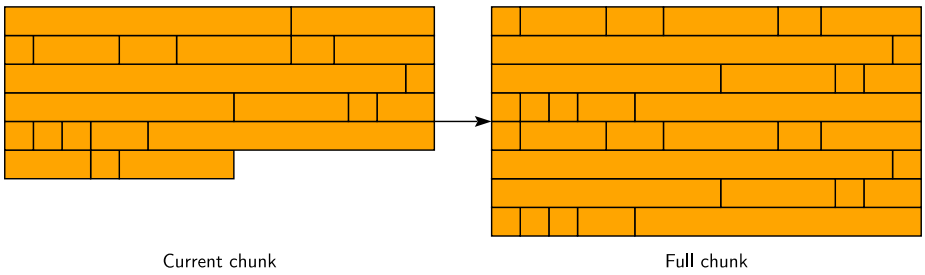


Figure 5: Bump allocation in chunks

Bump allocation is used because it is tight (removes *internal* fragmentation) and cheap. Early return of memory is uncommon, which will lessen the problem of external fragmentation of bump allocation, and the buddy allocator will leave no external fragmentation when the classloader is purged. For the less common cases when memory is freed before class loader destruction, the memory will be placed in `FreeBlocks`.

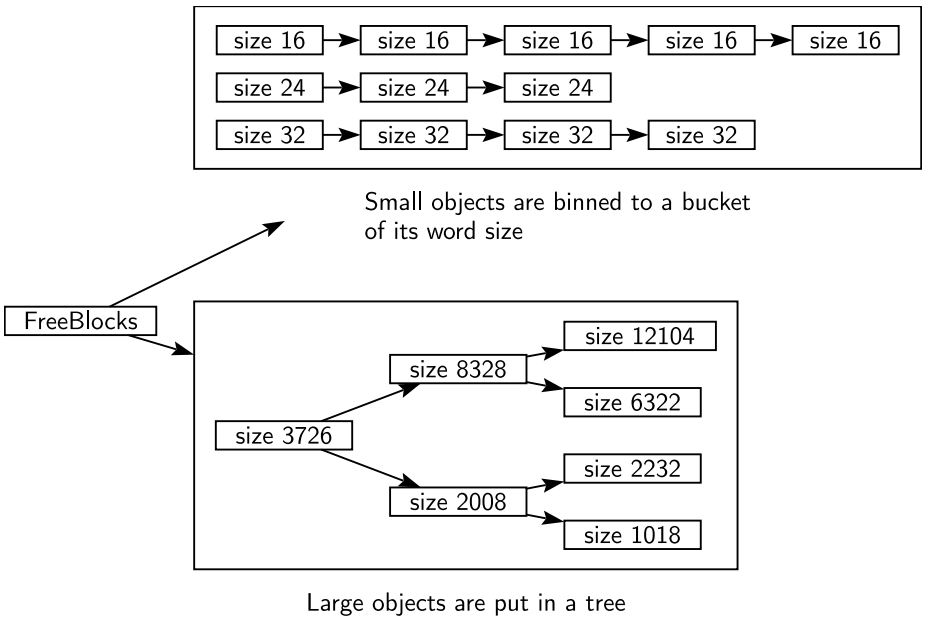


Figure 6: `FreeBlocks` (storage of freed allocations)

This "free list" is implemented as a lookup table for small binned sizes, and as

a tree for larger sizes. When the class loader is destructed, free chunks will be returned and merged back by the buddy allocator reducing fragmentation.

Committing and releasing memory

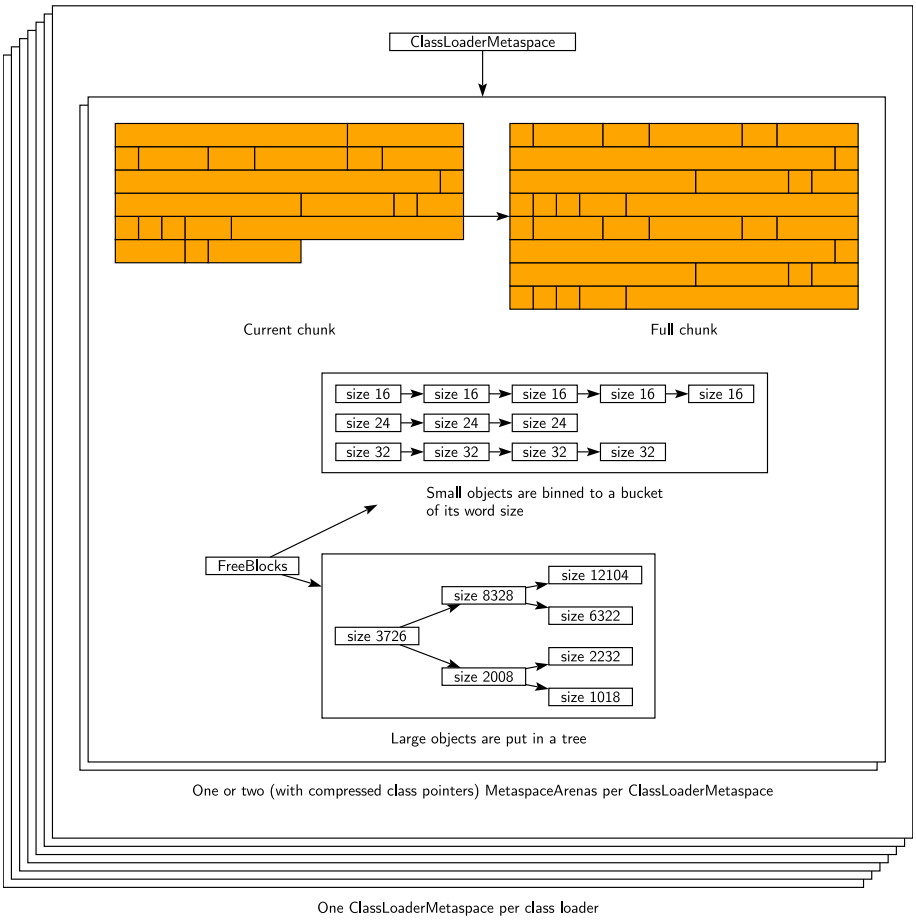
Each `VirtualSpaceNode` contains a bitmap that keeps track of what memory is committed. Every bit represents a commit granule. Even if a big chunk is allocated, the memory is committed lazily on bump allocation, and thus, memory usage is kept tight to a commit granule.

One of the improvements to the new Metaspace is how committing memory is now handled on a granule level; this is much tighter than the previous predetermined chunk size. This in combination with fragmentation reductions will result in less memory usage by the JVM process.

As objects are always allocated by chunks belonging to a specific class loader (by the buddy allocator), memory allocations between different class loaders never interleave (within a chunk). When a class loader is purged, most memory of the chunks can be returned to the operating system as class loader data from other class loaders can not interfere.

Further reading

If you want a better and more complete picture of Metaspace, you should read the much more detailed review guide of the Metaspace written by the implementer Thomas Stüfe, or visit his blog for video presentations, and other topics. There is also the Metaspace wiki at OpenJDK.



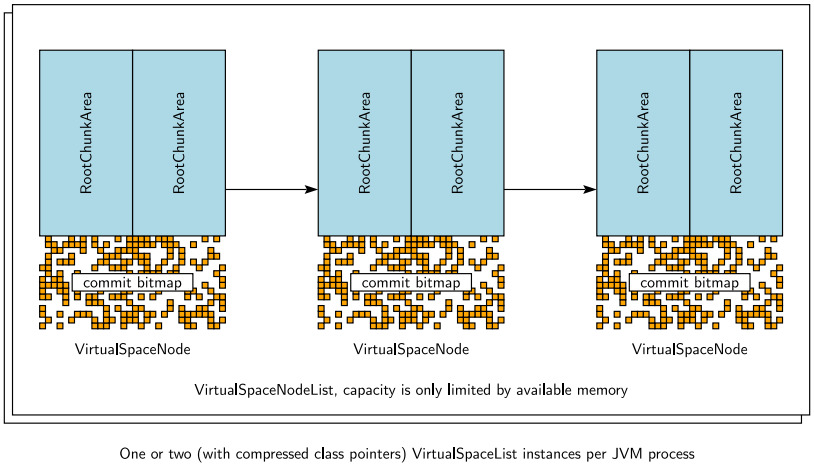


Figure 7: All parts interacting