Taming Metaspace

Thomas Stüfe, SAP
Feb 01, 2020

Follow us

www.sap.com/contactsap

THE BEST RUN SAP®

Basics

Metaspace

- Metaspace contains class metadata
  – Klass, Constant Pool, Method, Annotations, Bytecode etc.
  – JIT Counters

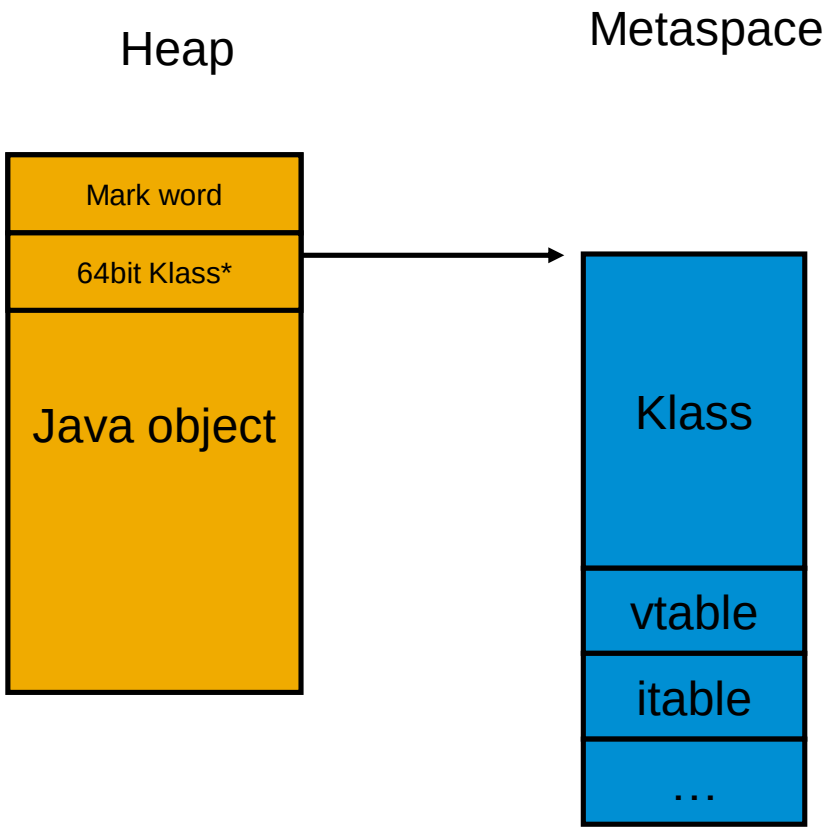
- JDK8: PermGen Removal (Java Heap -> Native Memory)
  – JEP 122: “**JEP 122: Remove the Permanent Generation”**

- SAP involvement:
  – JDK11: **JDK-8198423**, chunk coalescation patch to fix chunk size starving
  – Analysis tools: **jcmd VM.metaspace**
  – many smaller fixes/cleanups

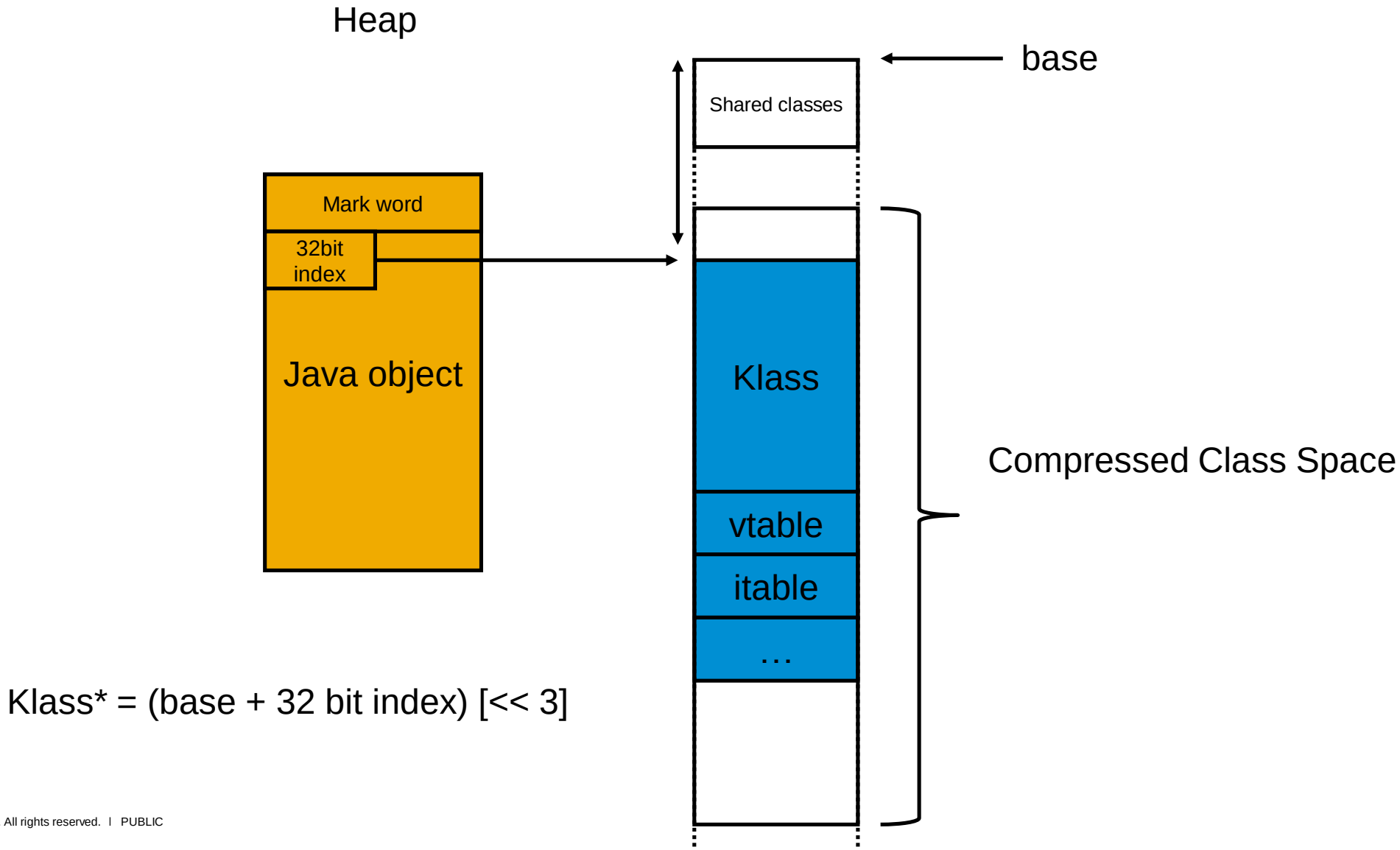
Metadata lifecycle

- Metadata are usually allocated when classes are loaded
- Metadata are freed in bulk when associated loader is collected
  – may be deallocated earlier (class redefinition, load errors etc) but that’s uncommon
- Bulk delete:
  – No need to track individual allocations
  – Arena based allocation sufficient

Compressed Class Space (1)

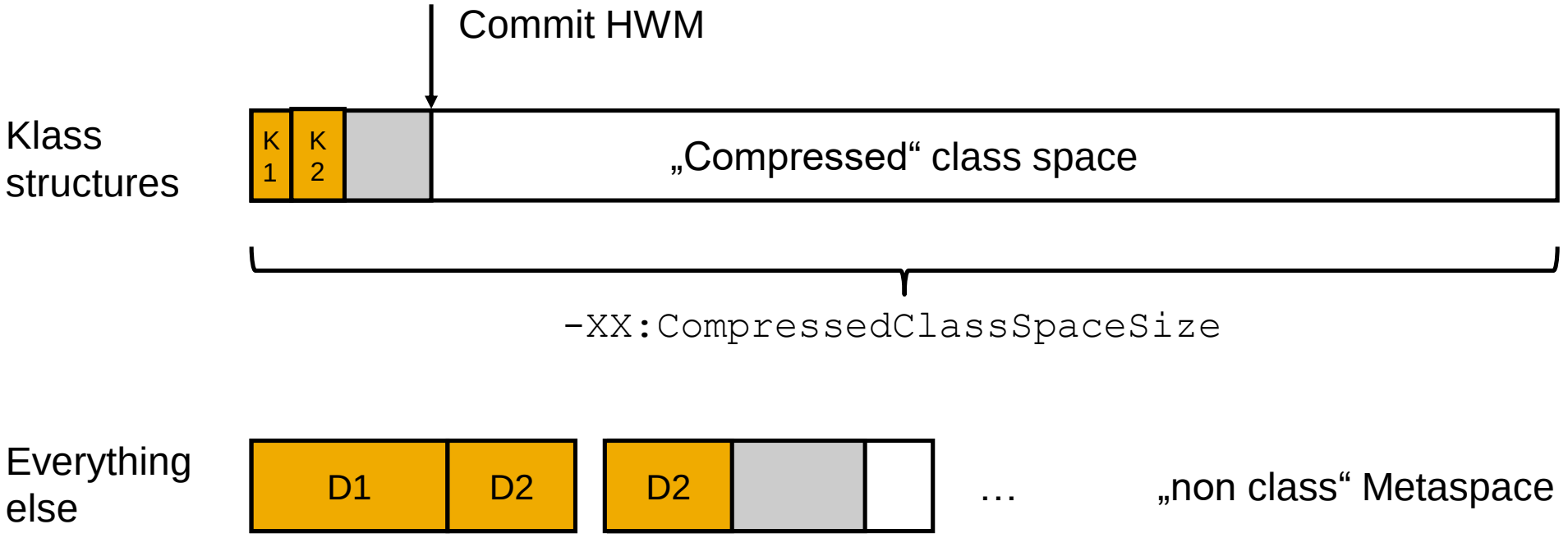


Compressed Class Space (2)



Klass* = (base + 32 bit index) [<< 3]

Metaspace has two parts



Sizes per class:

- ~1K Klass (500+ … 500K)
- ~6K non-class (~2K … xxK)

Limits

- **CompressedClassSpaceSize**:
  - Reserved size of compressed class space. Max 3G.
  - Has to be specified. If omitted, defaults to 1G (~ 1 million classes)
- **MaxMetaspaceSize**
  - Limits sum of all committed space (class + nonclass)
  - Default infinite.

- Sizing:
  - Undersizing can hurt (OOMs, GCs)
  - Oversizing is usually not a problem. When in doubt, keep the defaults.

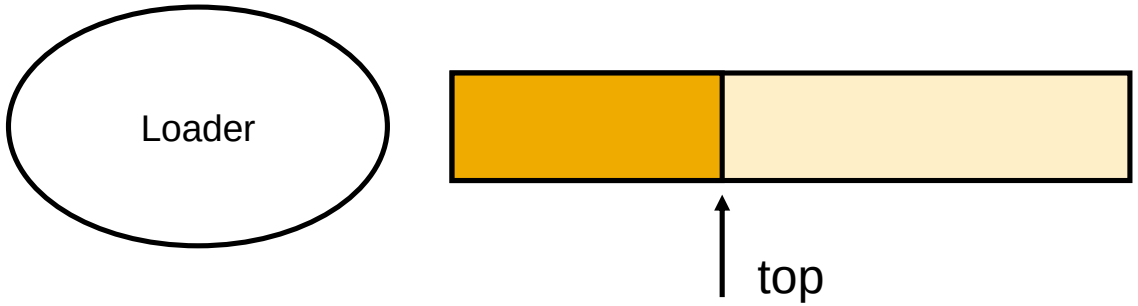
Metaspace induced GCs

- GC is blind to Metaspace consumption, so we need to periodically check if classes can be collected
- When sum of committed Metaspace reaches a threshold, further allocations are delayed until a GC is done. Maybe we can collect some loaders and release their Metaspace?
  – Threshold may go up or down depending on how the GC went
- We also attempt a GC before throwing a Metaspace OOM
- *MetaspaceSize* sets the initial threshold value
  – Set to a large value (e.g. max) to disable threshold

Current implementation

Current implementation (1)

(much simplified)



- Loader owns a chunk of memory.
- Allocates from it via pointer bump.
  - Remember: we do not need to track individual allocations for freeing.

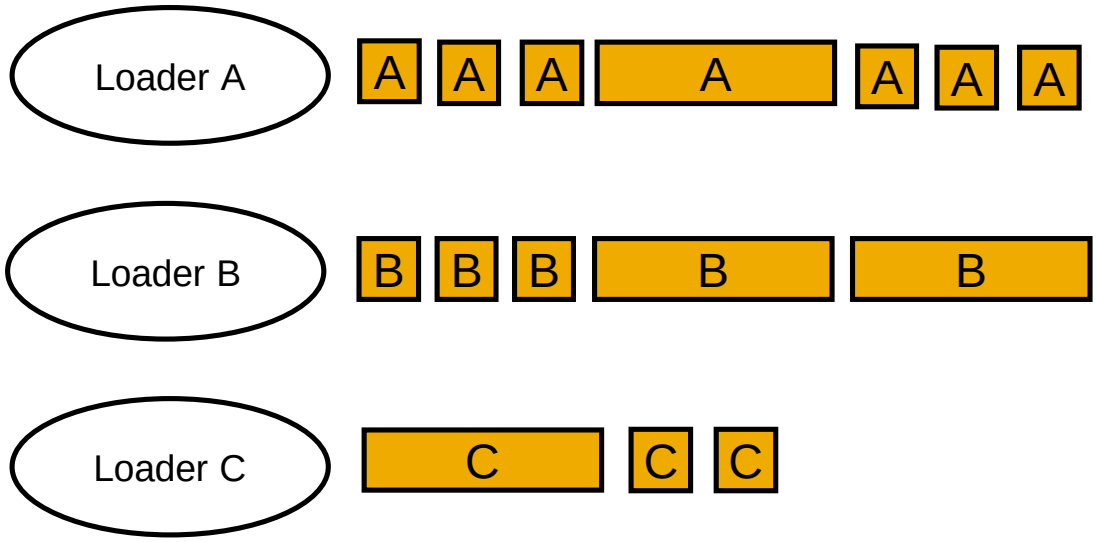
Current implementation (2)

(much simplified)

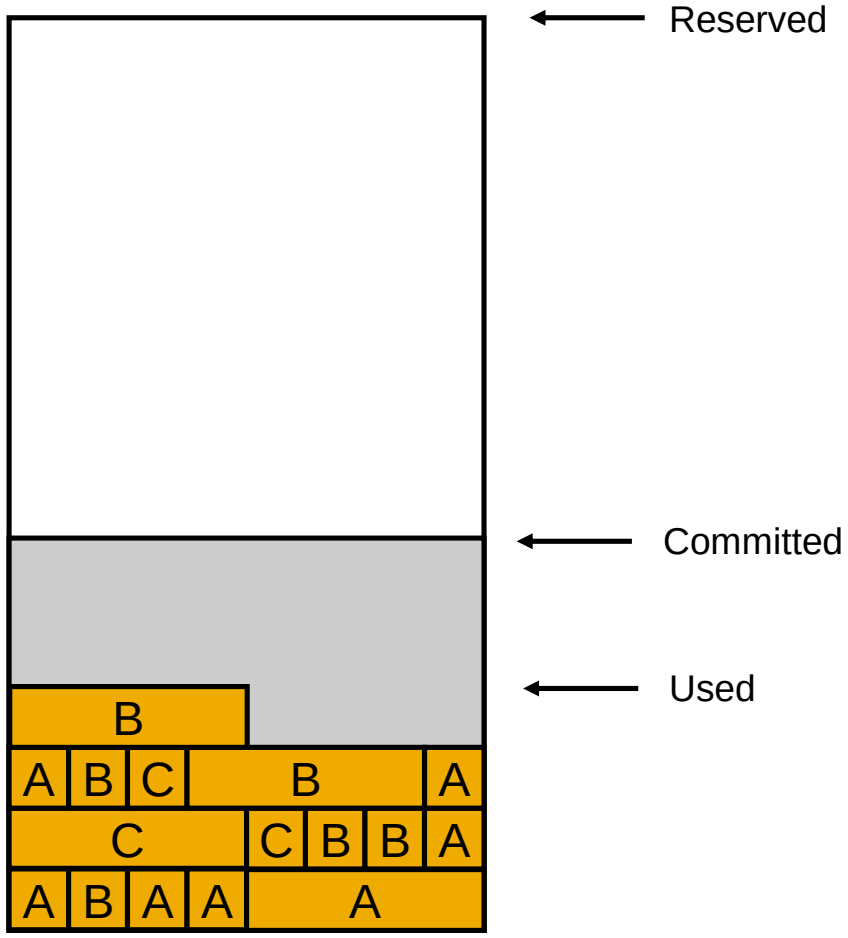
- If chunk is used up, Loader aquires a new one from the metaspace allocator.
- Retired chunks are kept in list
- Leftover space is kept for later reuse

Current implementation (3)

(much simplified)

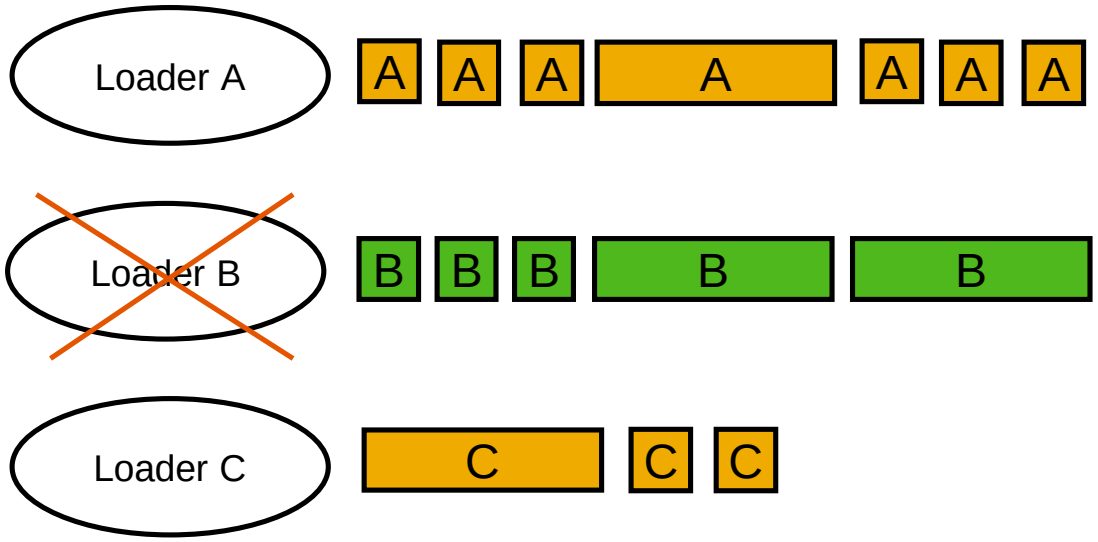


Chunks are carved from metaspace memory as they are allocated.

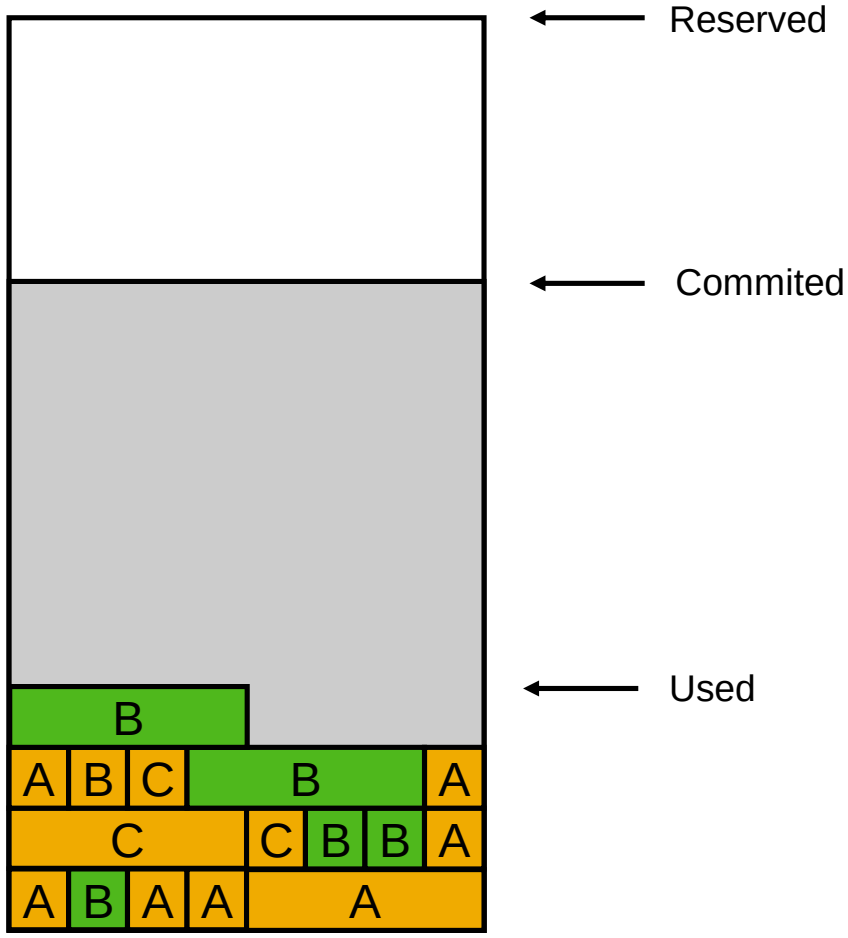


Current implementation (4)

(much simplified)

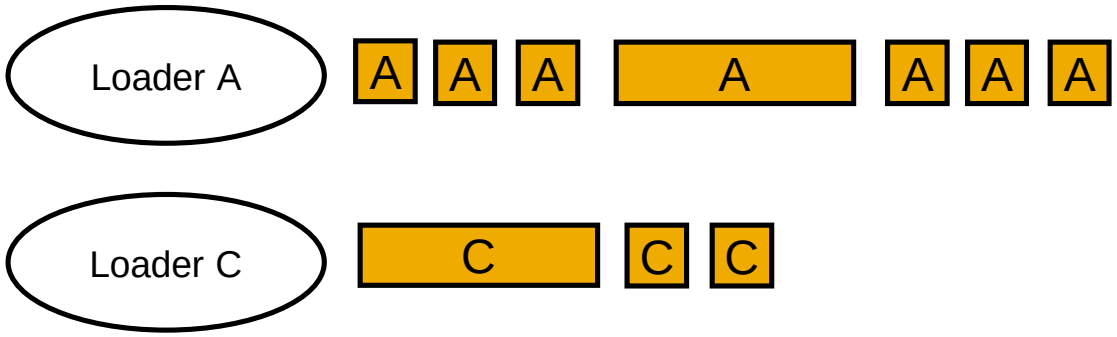


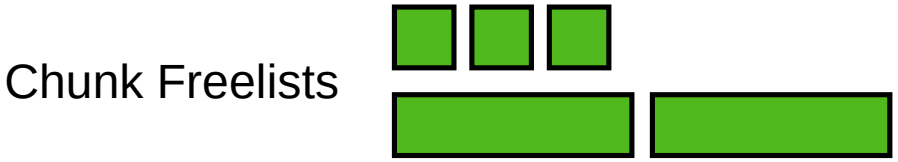
When a loader dies, its chunks are marked as free…



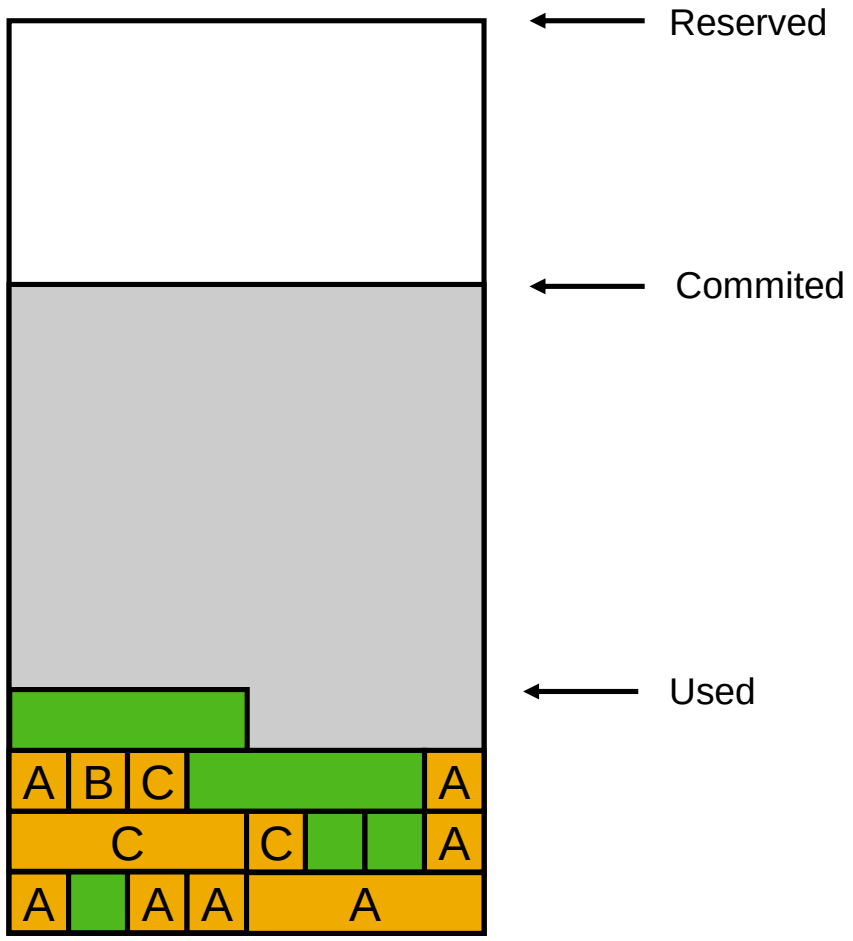
Current implementation (5)

(very much simplified)





…and added to global freelists, sorted by size.



Monitoring with `jcmd VM.metaspace` (since JDK11)

- Detailed analysis of Metaspace occupancy
- Usage stats, chunk statistics and -geometry, freelists, …
- Show all loaders and/or all loaded classes and their space consumption
- Detailed summary waste section
- Help is your friend (`jcmd help VM.Metaspace`)

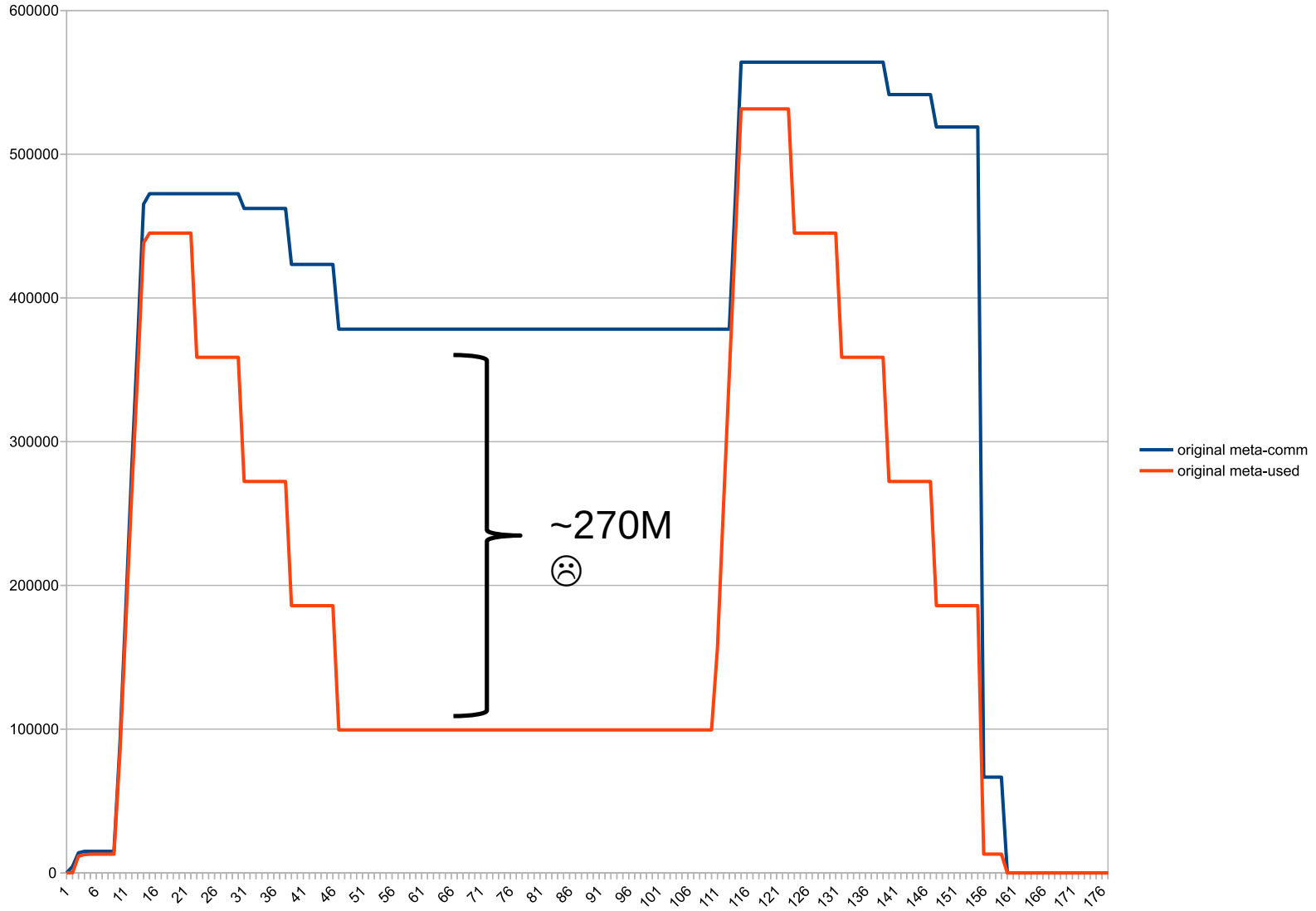
Problems with the current implementation

- Freelists can get huge.
  – We have seen used:free ratios of 1:3 and worse
  – =>Metaspace is not really elastic.

- Intra-chunk waste
  – At some point loader typically stops loading classes; remaining chunk space is wasted
  – Worse with many tiny loaders (reflection delegator classes, lambda anonymous classes)

- Code bloat
  – Expensive to maintain.
  – Code base grew over time and has gotten overly complicated.

Huge freelists: Committed vs used space, after class unloading



Huge Freelists (`jcmd VM.metaspace` output)

```
jcmd 27265 VM.metaspace

27265:
…

Waste (percentages refer to total committed size 373,48 MB):
              Committed unused:    280,00 KB ( <1%)
        Waste in chunks in use:      2,45 KB ( <1%)
         Free in chunks in use:      6,34 MB (  2%)
     Overhead in chunks in use:    186,75 KB ( <1%)
                 In free chunks:    269,56 MB ( 72%)
Deallocated from chunks in use:    998,98 KB ( <1%) (1763 blocks)
                       -total-:    277,33 MB ( 74%)
```

Reimplementation

Basic idea

- Uncommit chunks in freelists

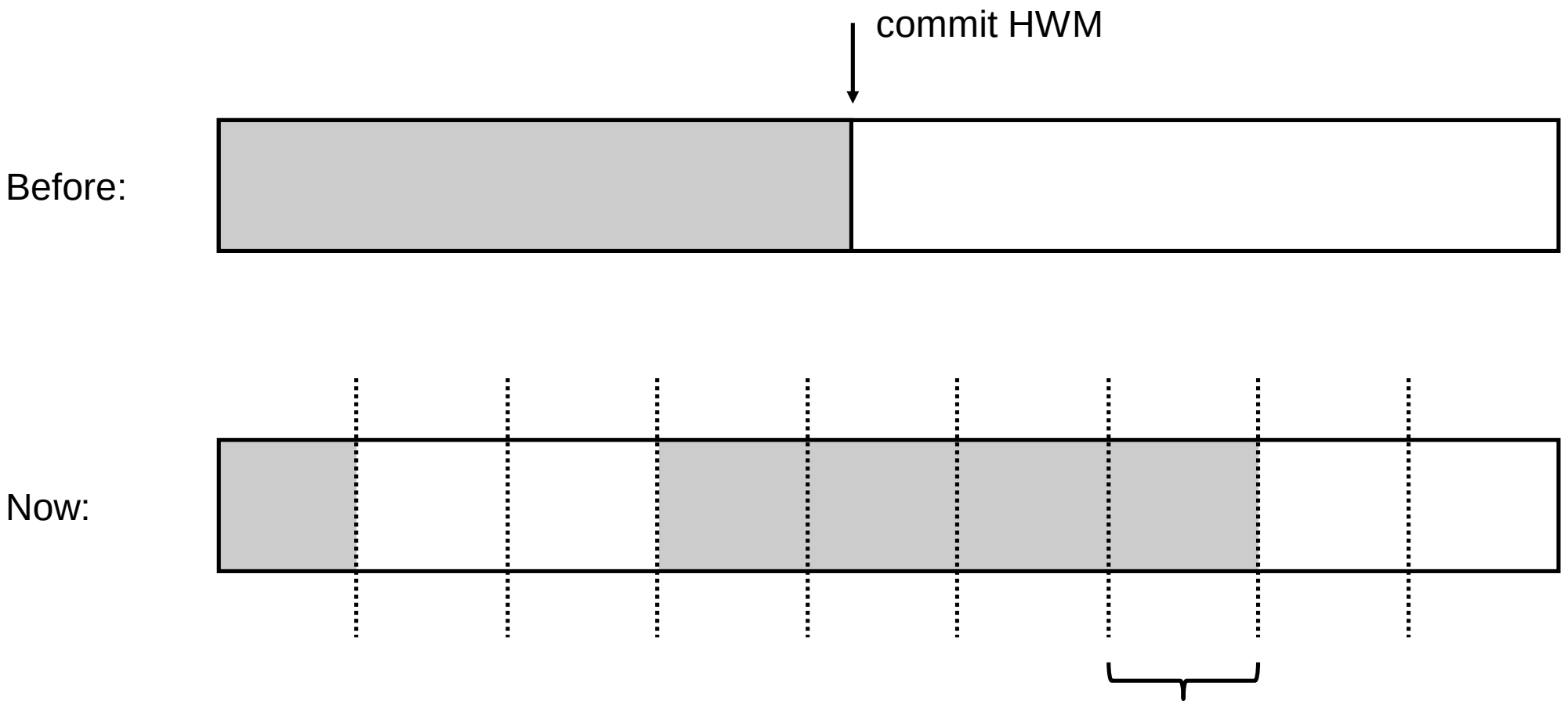
- Delay committing chunks until they are actually used
  – Partly commit them piece wise (like a thread stack)
  – Removes the penalty of handing out large chunks to class loaders

Concern: keep number of vm areas low

- (Linux): we decommit with mmap(MAP_NORESERVE) && mprotect(PROT_NONE)
  - May create a new vma (or, two)
  - Kernel keeps vma structures in list and tree
  - Too many of them may affect vma lookup
  - And we may hit process limits

- Hence: avoid fine-grained decommits

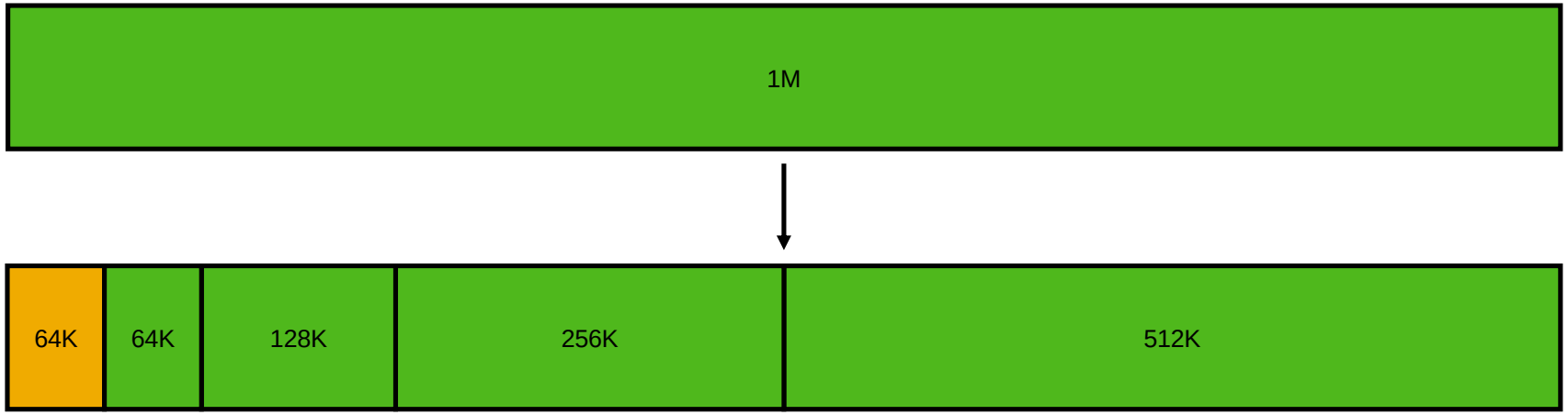
Commit granules



Pow 2 based buddy allocator for chunks

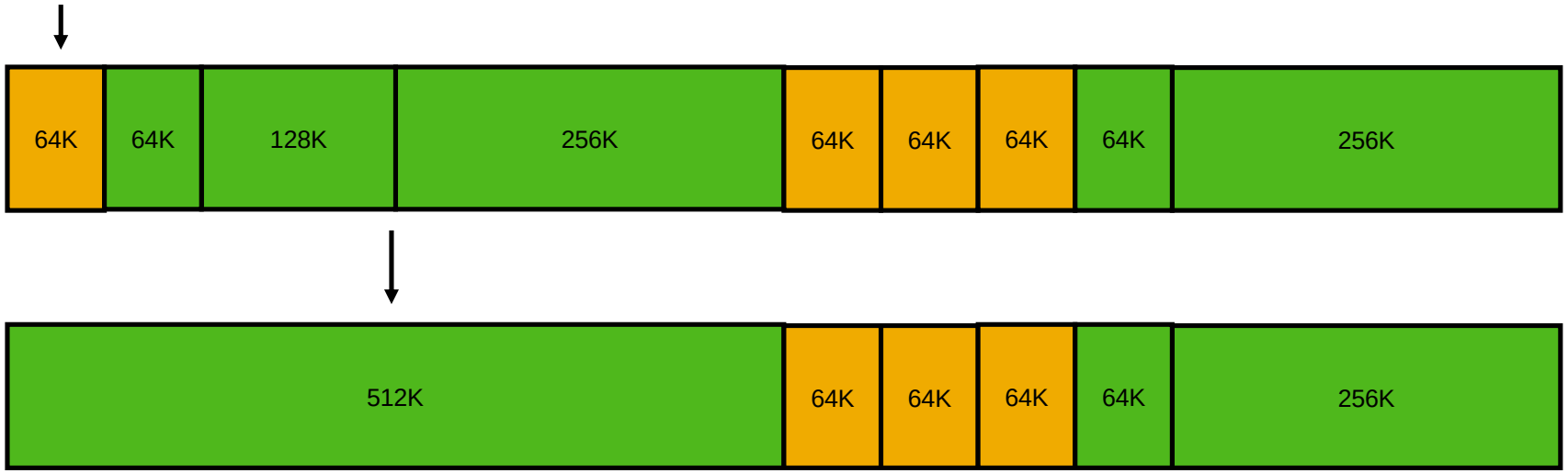
- Power 2 based buddy allocation scheme
- Chunks sized from 1K … 4M in pow2 steps
- Dead simple to split and merge.
- Low external defragmentation -> Leads to larger free contiguous areas.
- Standard algorithm widely known

Buddy allocator: Allocation



- Remove chunk from freelist
- Optionally split until desired size is reached
- Return result chunk; put splinter chunks back to freelist

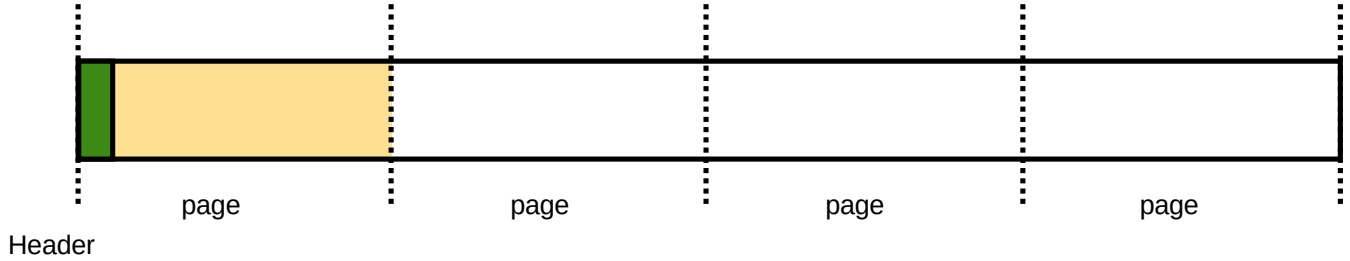
Buddy allocator: Deallocation



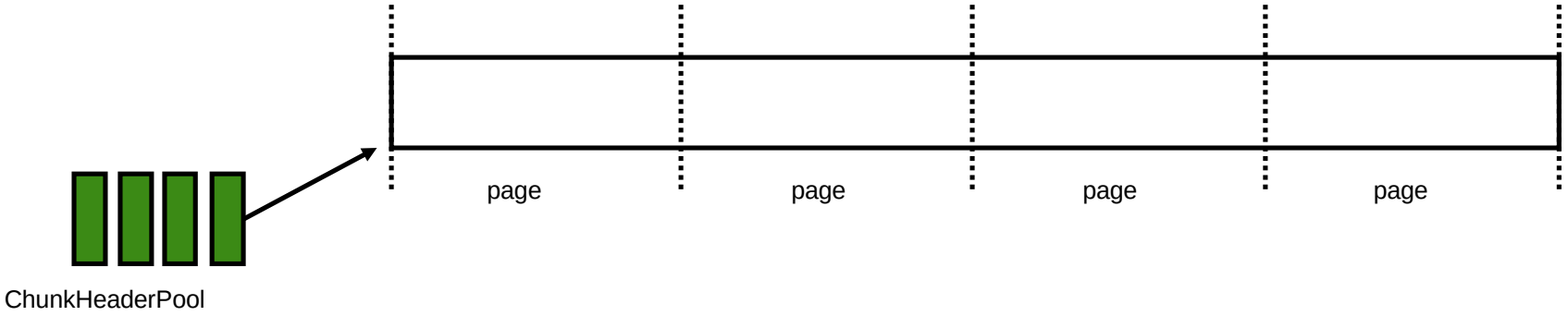
- Mark chunk as free
- If buddy chunk is free and unsplit: remove from freelist and merge with chunk
  – Repeat until root chunk sized reached or until buddy is not free
- Return result chunk to free list

Chunk headers needed to go

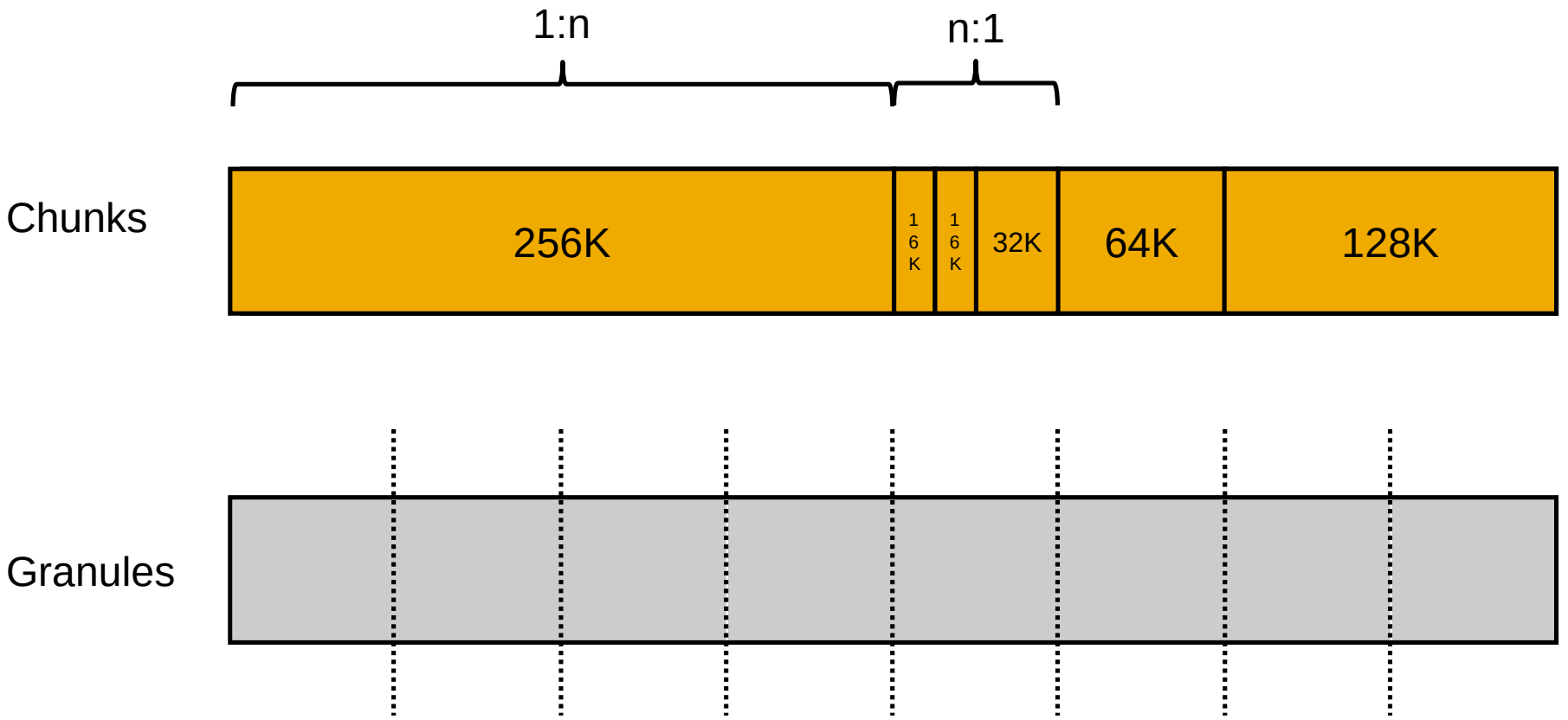
Before



Now

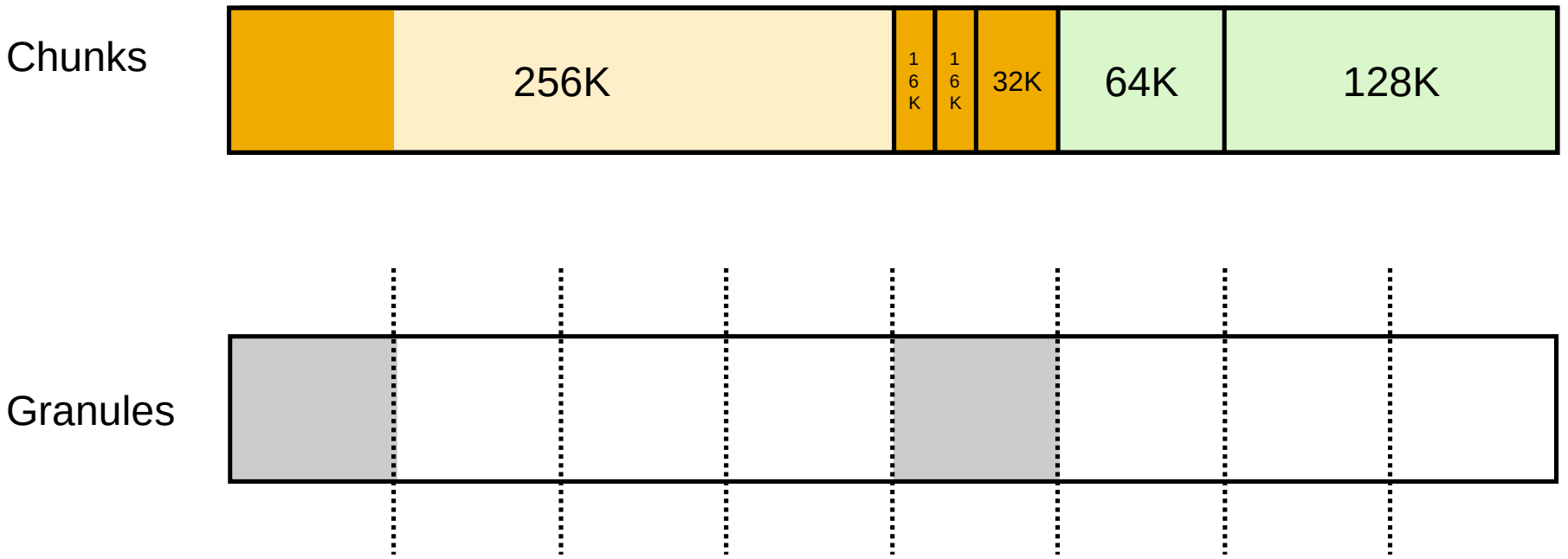


Granules and chunks



- A larger chunk can span multiple granules (1:n)
- Multiple small chunks can cover a single granule (n:1)

Granules and chunks

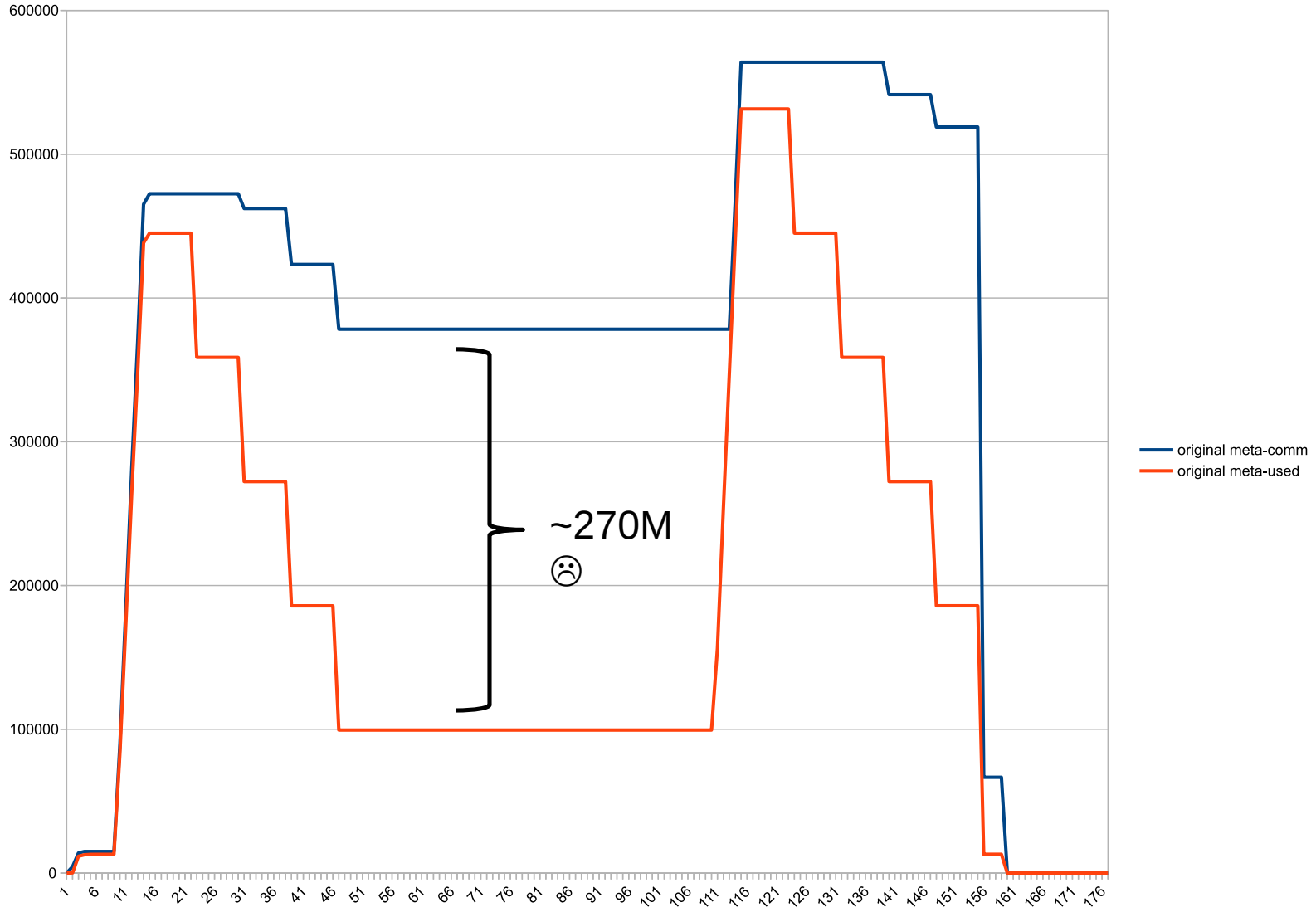


- Free chunks spanning 1+ granules can be uncommitted
- A chunk spanning >1 granules can be committed on demand

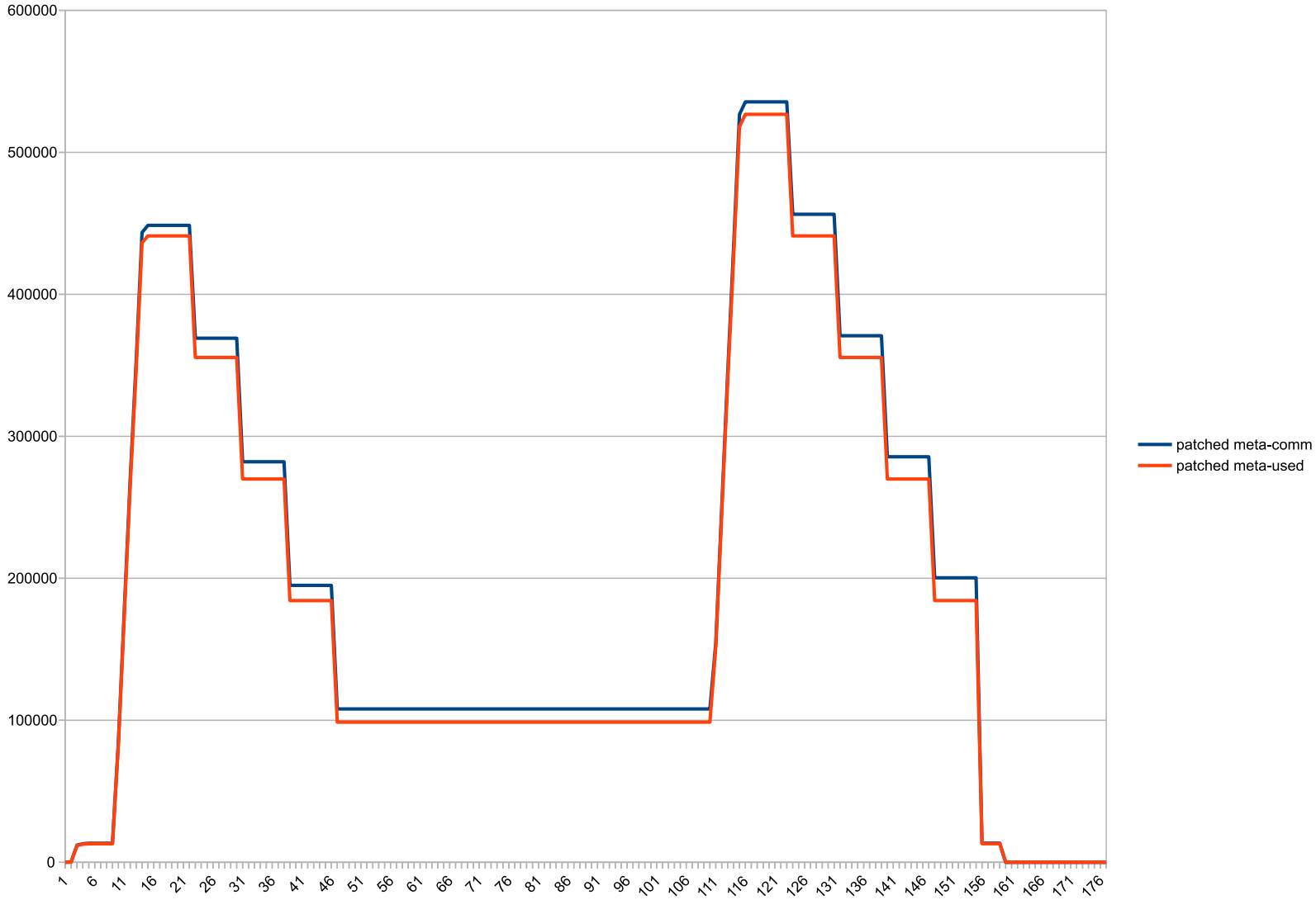
Other Changes

- Got rid of humongous chunks :)
- Got rid of occupancy map
- Better leftover management
- Chunks can now often grow in-place
  - Saves overhead and reduces intrachunk waste
- Code is cleaner and more maintainable; better separation of concerns and testability.

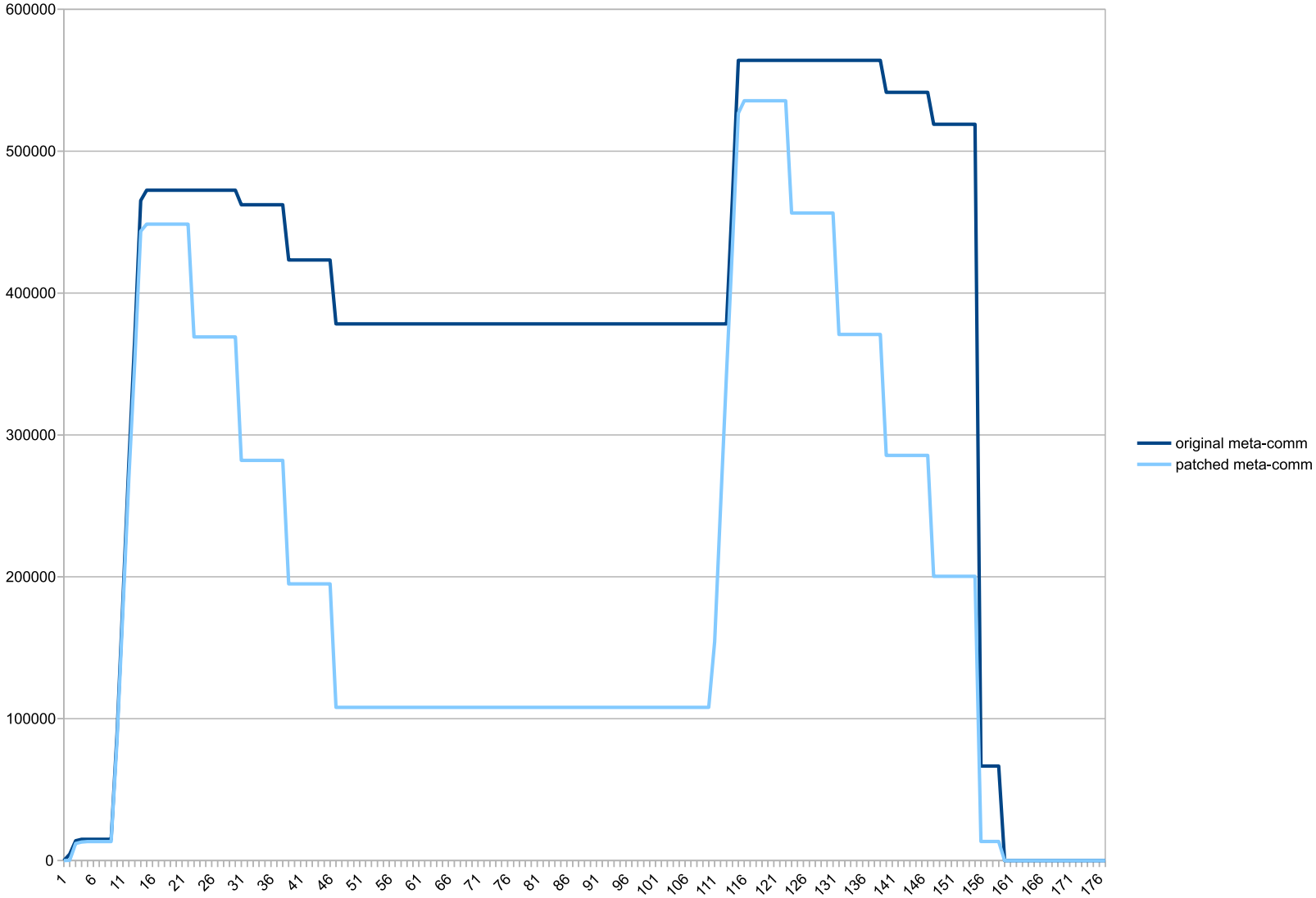
Result: Committed vs used, Stock JDK14



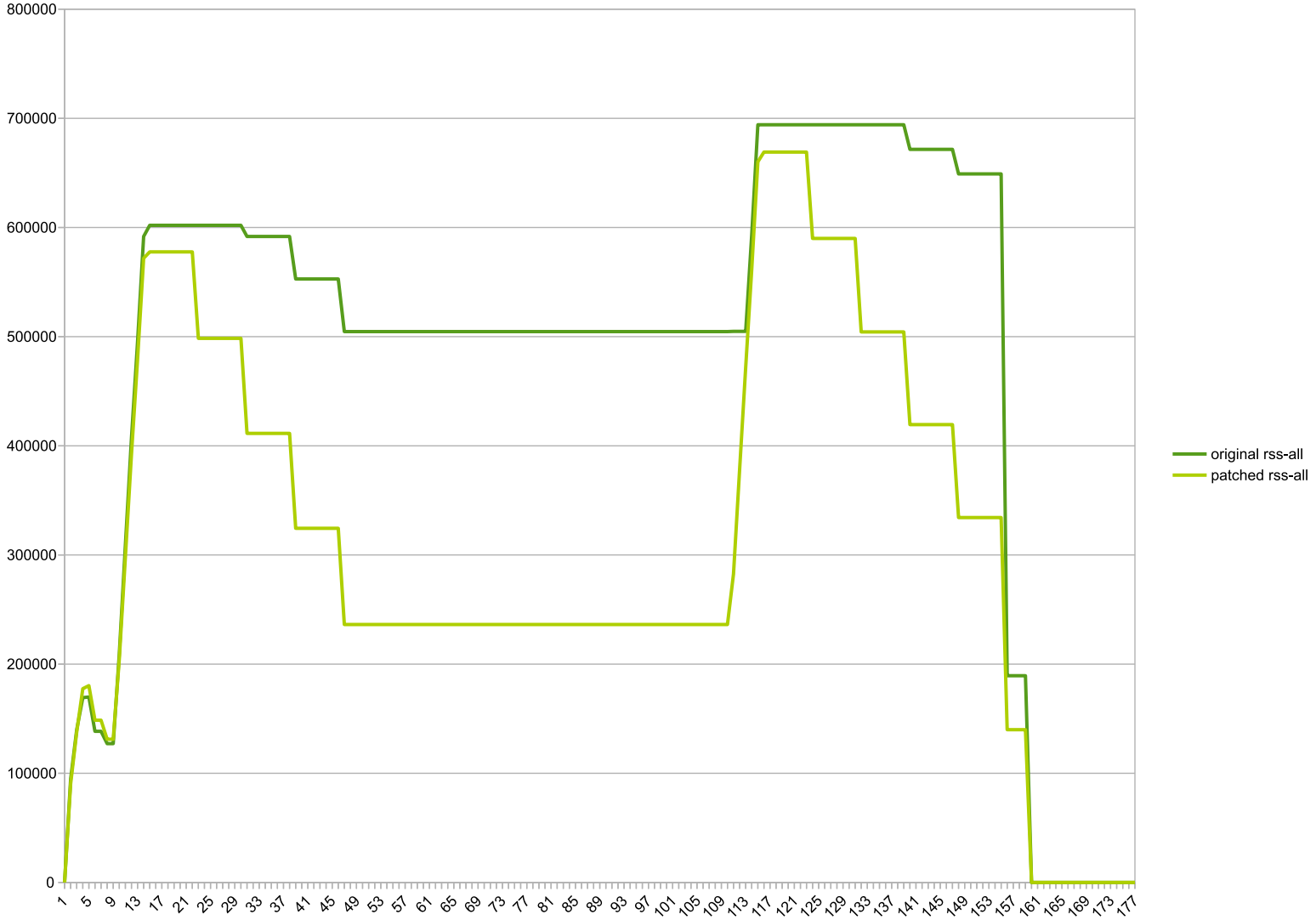
Result: Committed vs used, Patched JDK14



Result: committed Metaspace, Stock vs Patched VM



Result: RSS, Stock vs Patched VM



Modest decrease in consumption beyond class unloading:

- Wildfly standalone, after startup: 61m -> 54m
- Eclipse CDS (OpenJDK project open, after C++ indexer ran): 151m -> 131m
- Minecraft 1.14: 74m -> 70m

Show me the code

- http://hg.openjdk.java.net/jdk/sandbox/
  – branch "stuefe-new-metaspace-branch“

- ~20kloc

How do we go from here?

- Patch is stable. Needs more TLC but it works.

- JDK15?
  – Very difficult to bring such a large patch upstream :(
  – A JEP exists in Draft state (“Elastic Metaspace”: https://openjdk.java.net/jeps/8221173)

- A good candidate for backporting!
  – Would make a lot of sense in 11/8
  – Large patch but Metaspace is quite isolated. Should not be too much of a hassle.

Thank you.

Contact information:

Thomas Stüfe
@tstuefe
thomas.stuefe@sap.com
stuefe.de

Q/A