# Taming Metaspace

Thomas Stüfe, SAP
Feb 01, 2020

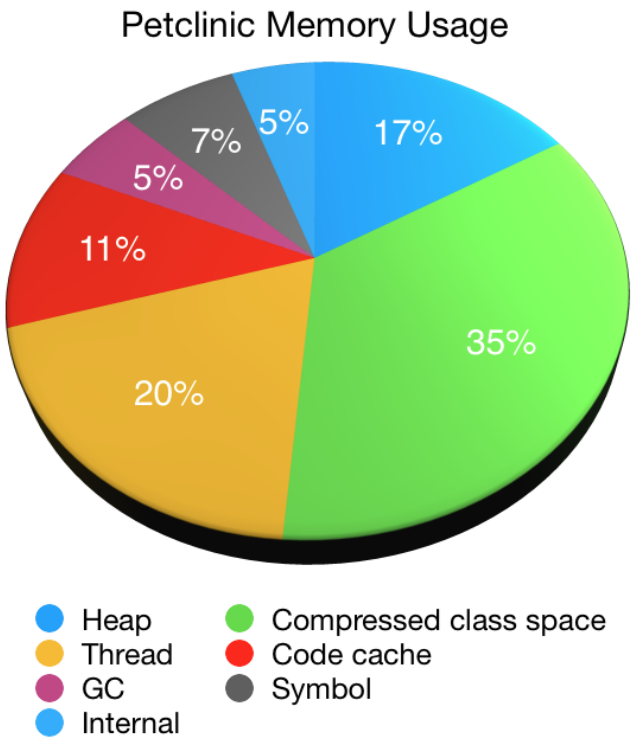
Follow us

**www.sap.com/contactsap**



THE BEST RUN SAP®

# Why do we care?




(Taken from
https://spring.io/blog/2019/03/11/memory-footprint-of-the-jvm

Note: Metaspace is mislabeled as Compressed Class Space)

Metaspace can consume a lot of memory.

Can we improve that? Yes, within reason.

# Basics

# Metaspace

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

- Used to live in Java Heap (Permanent Generation) pre JDK 8

- JDK 8: PermGen Removal -> Metaspace is born
  – Inspired by JRockit VM
  – JEP 122: “**JEP 122: Remove the Permanent Generation”**

- SAP involvement:
  – JDK 11: partial rework (chunk coalescation, **JDK-8198423)**
  – Analysis tools: **jcmd VM.metaspace, VM.classloaders**
  – many smaller fixes/cleanups
  – JDK 15 (?): rewrite

# Metadata lifecycle

- Metadata are usually allocated when classes are loaded

- All metadata a loader accumulated is freed in bulk after the loader has been collected
  - Exception: Metadata may be deallocated earlier (class redefinition, load errors etc) but that’s uncommon.
  - Deallocated space still belongs to the loader.

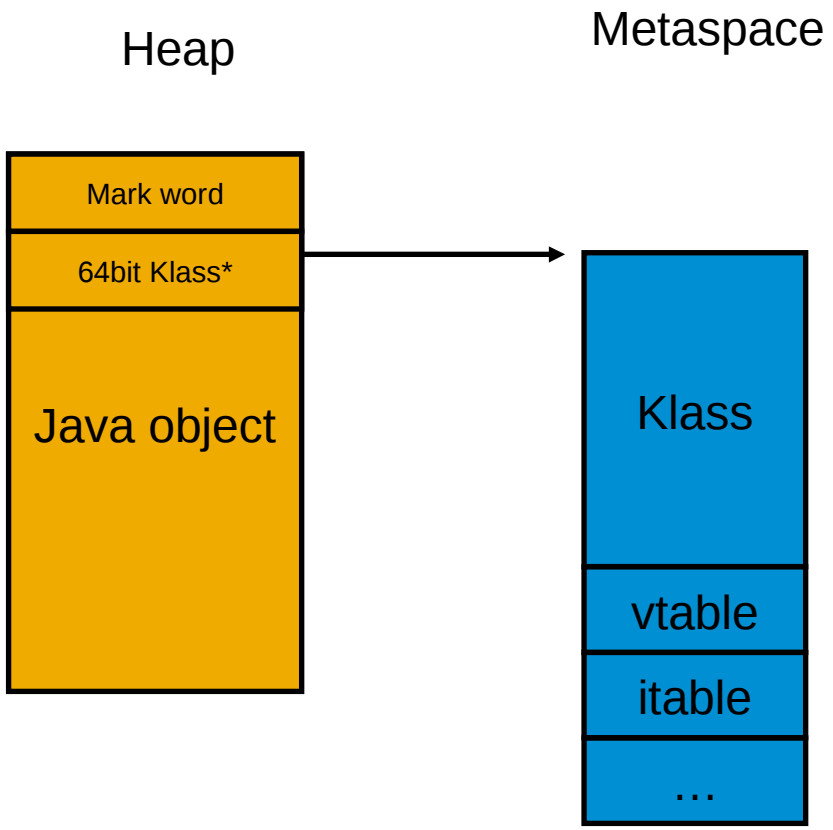
# Why write a customized allocator?

- Bulk delete allows arena style allocation!
  - No need to track individual allocations
  - Simple pointer bump allocation possible: cheap and allows tight packing
- We know the size distribution of typical allocations
- Case against malloc in particular:
  - CompressedClassSpace
  - Platform specific limitations (e.g. sbrk hits java heap)

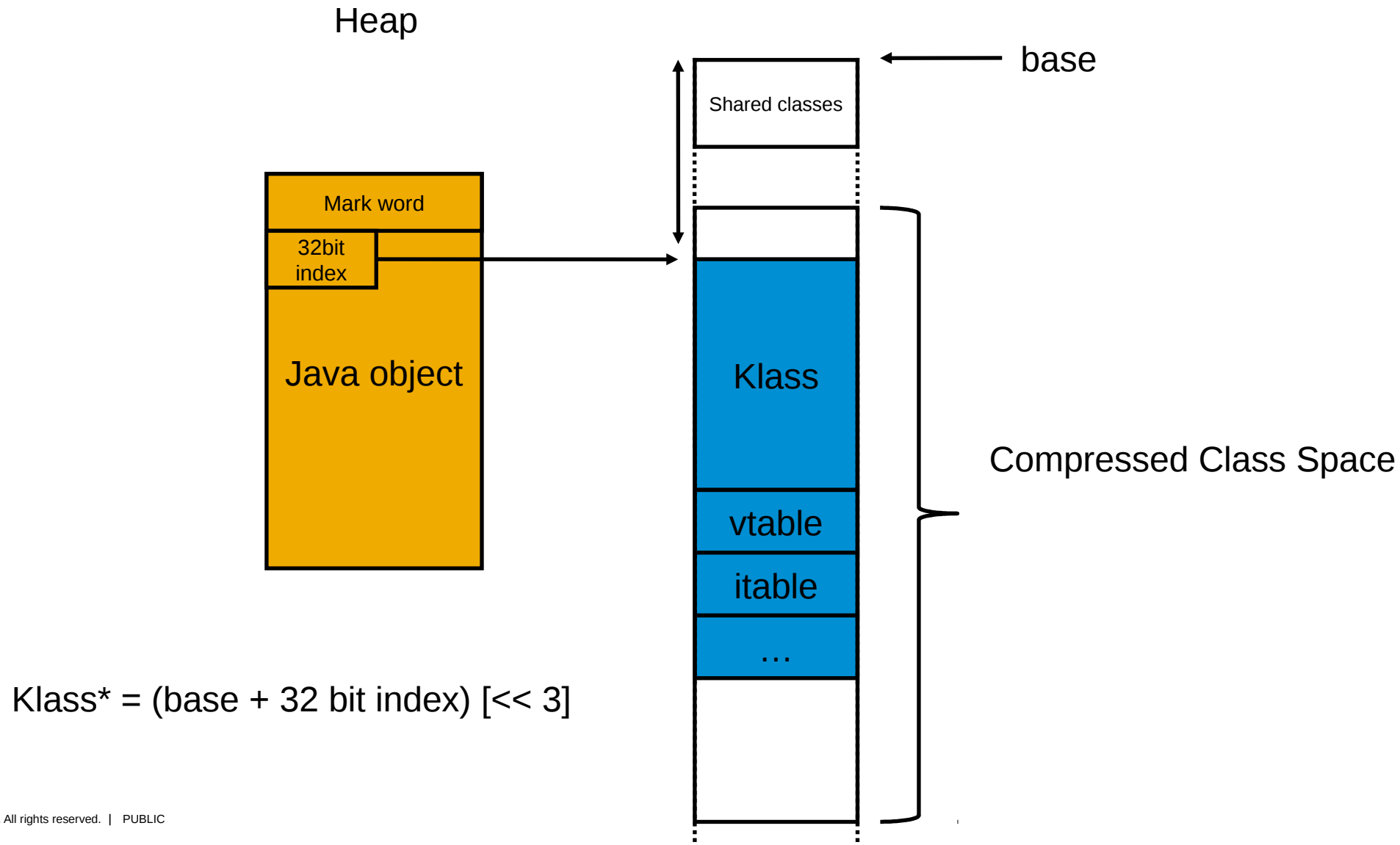
# Compressed Class Space

- Compressed Class Space is a (small) part of Metaspace
- Optimization for 64bit platforms
- Only on 64bit, if `-XX:+UseCompressedClassPointers` (on by default)

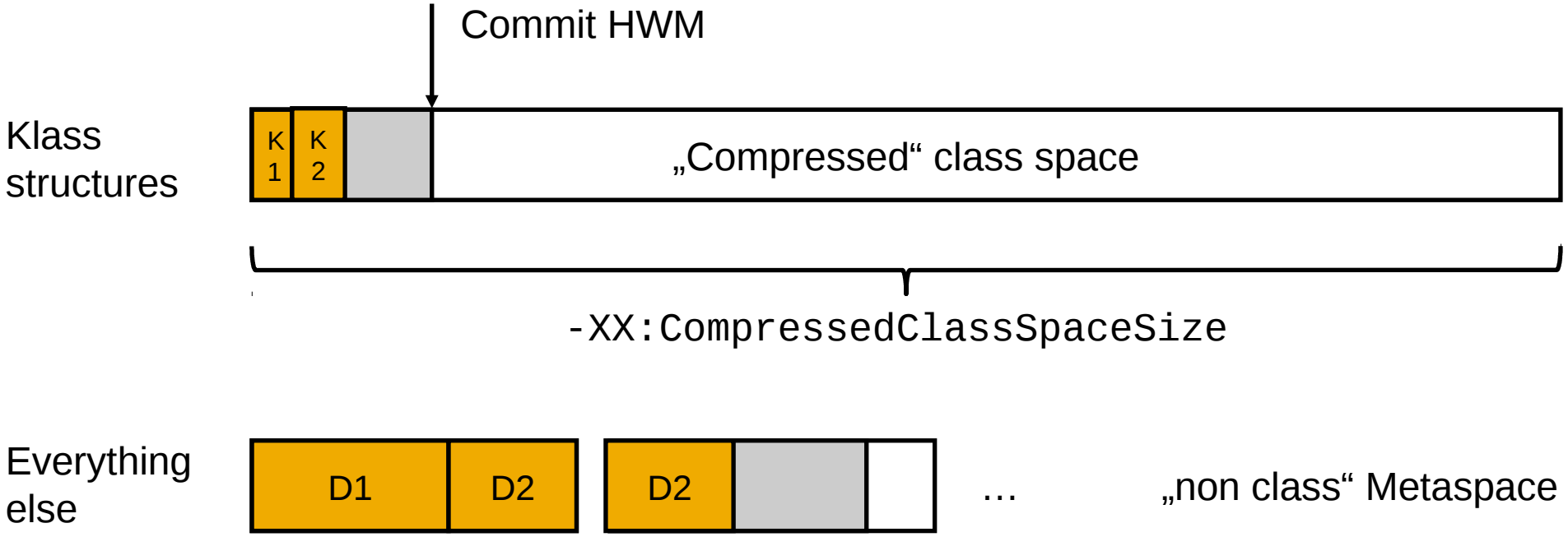
# Compressed Class Space



# Compressed Class Space



# Metaspace has two parts

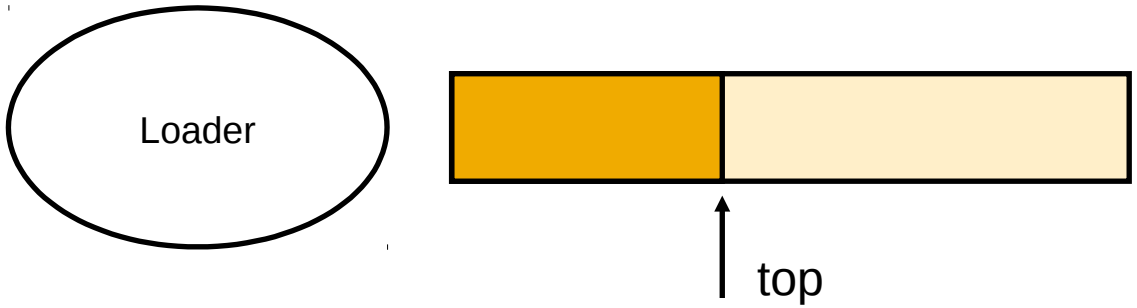



Sizes per class:

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

# 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.

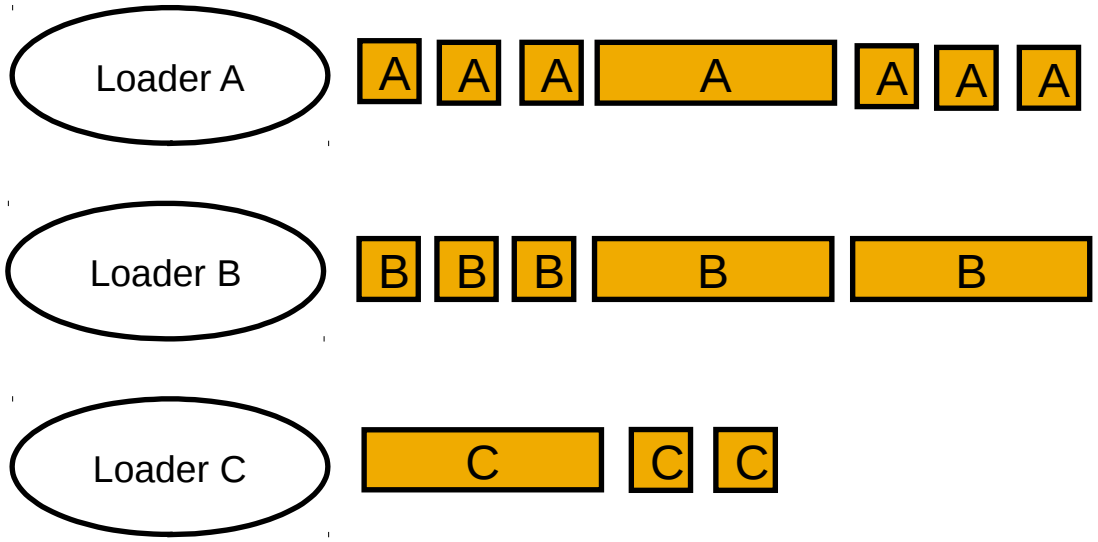
# Current implementation (2)

(much simplified)

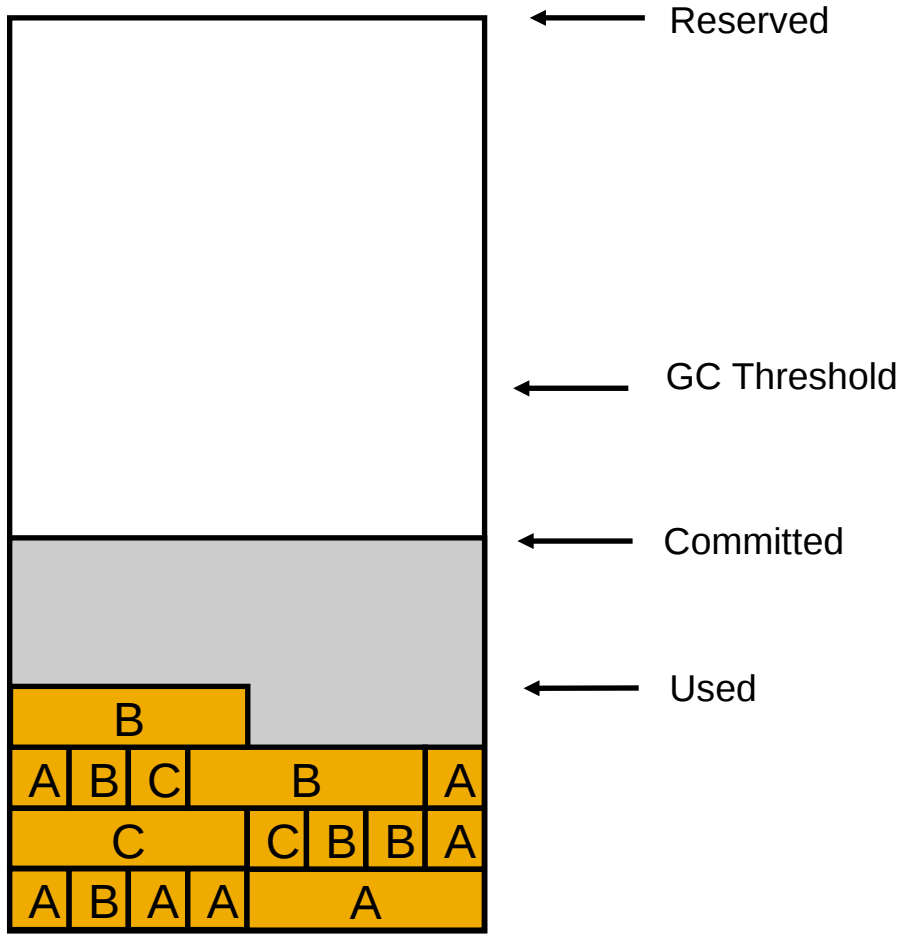

- If chunk is used up, Loader aquires a new one from the metaspace allocator.
- Retired chunks are kept in list
- Leftover space is kept for later reuse

# Current implementation (3)

(much simplified)

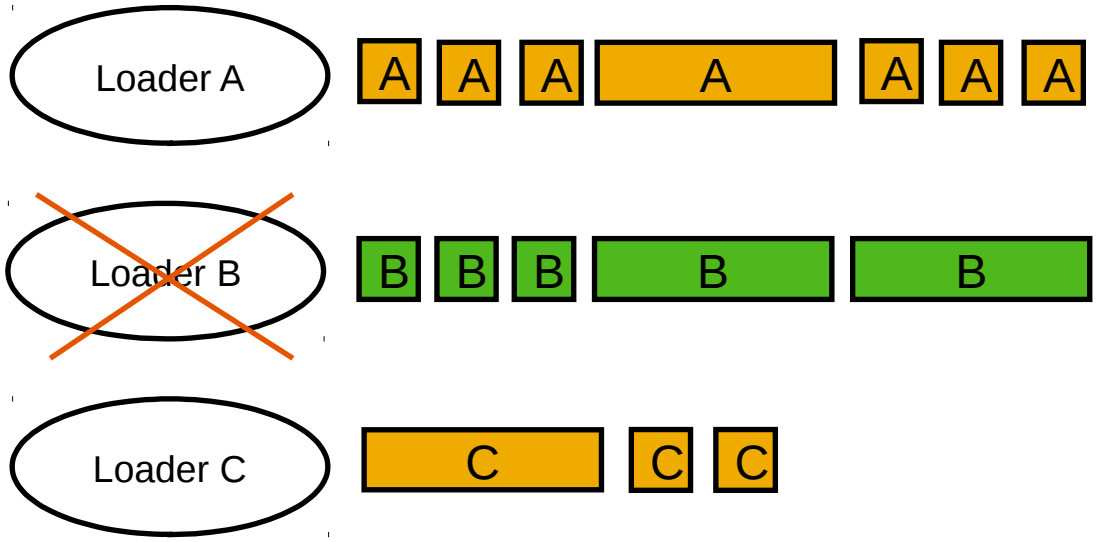



Chunks are carved from metaspace memory as they are allocated.

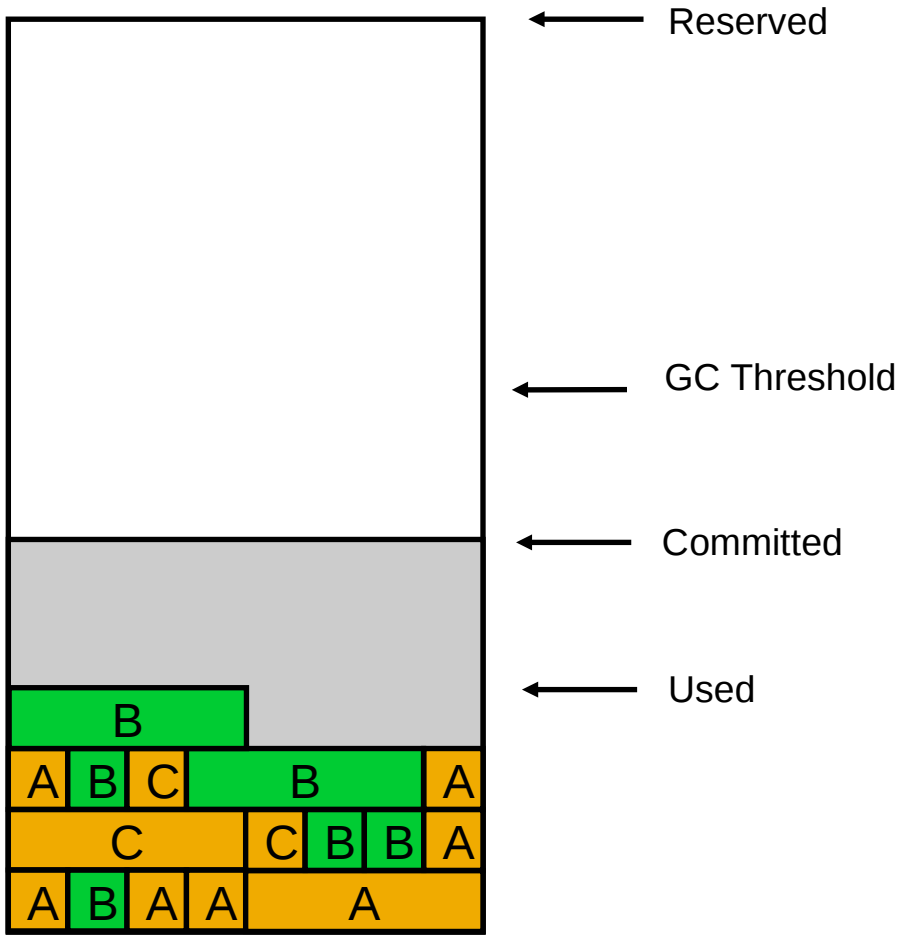


# Current implementation (4)

(much simplified)

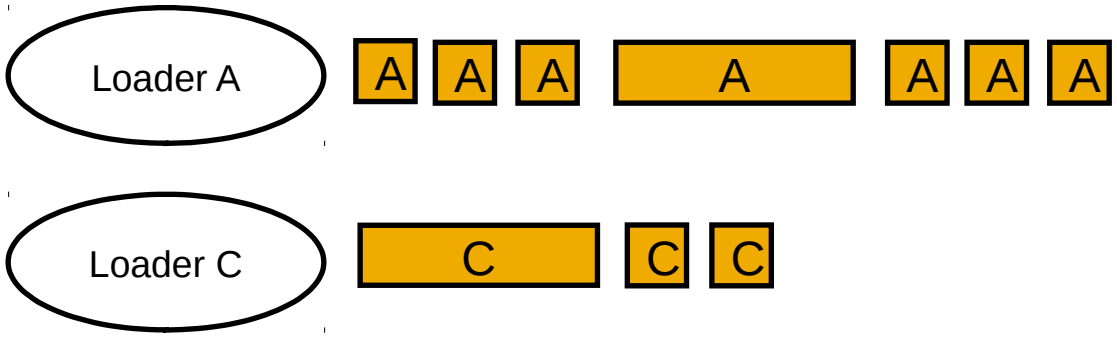



When a loader dies, its chunks are marked as free…

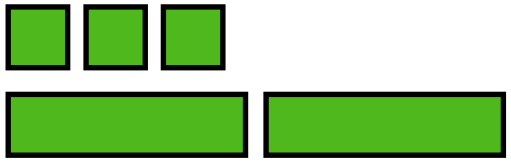


# Current implementation (5)

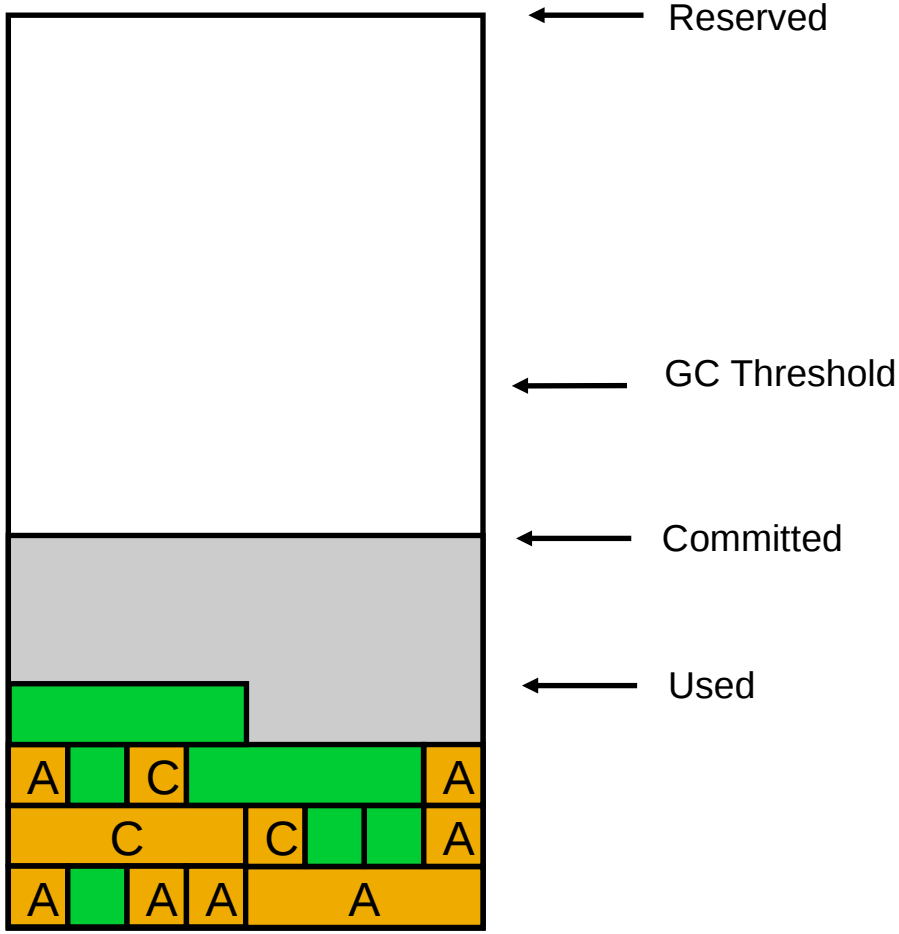
(very much simplified)




Chunk Freelists



…and added to global freelists, sorted by size.



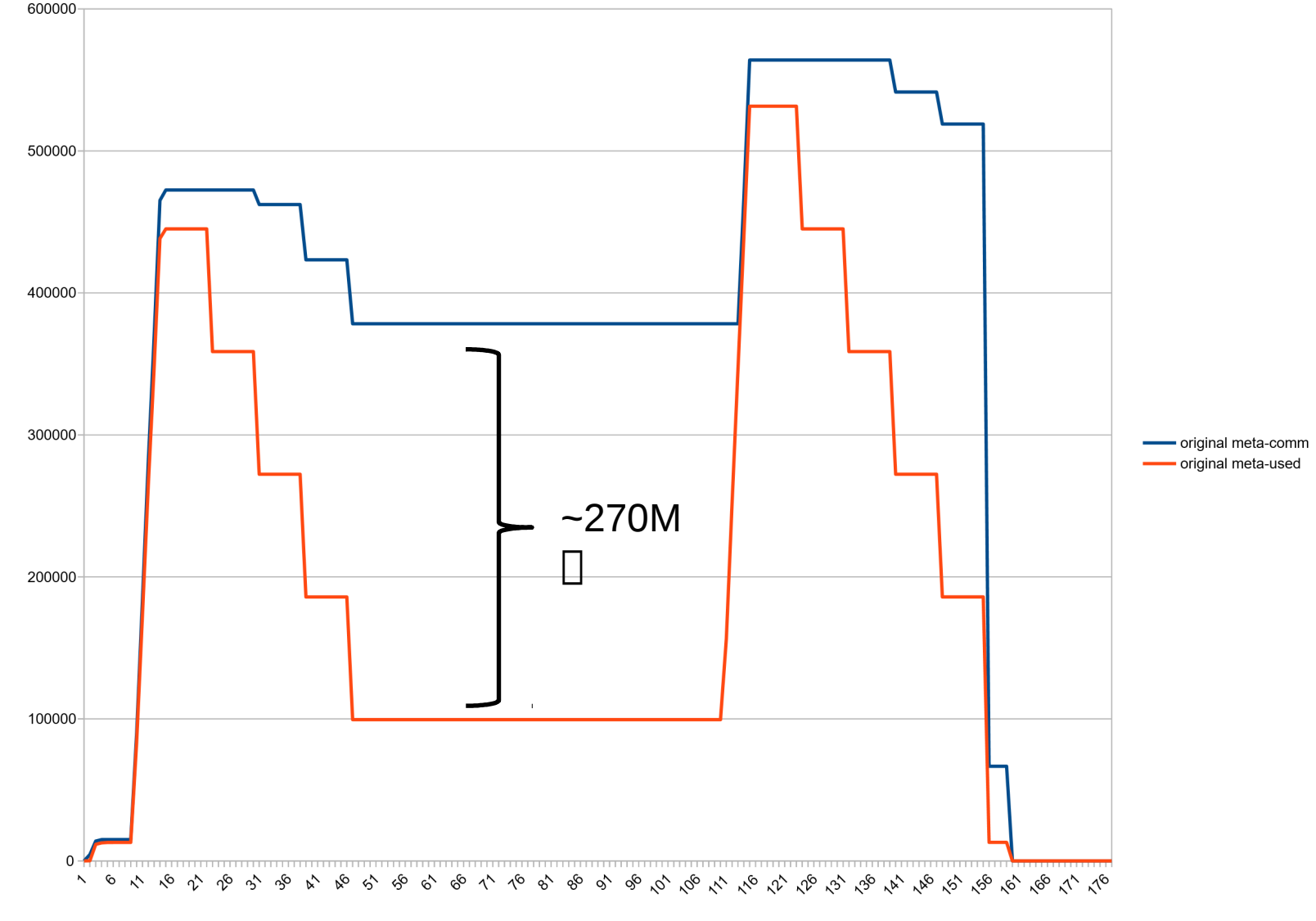
# Problems with the current implementation

- 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
    - Makes Metaspace more elastic

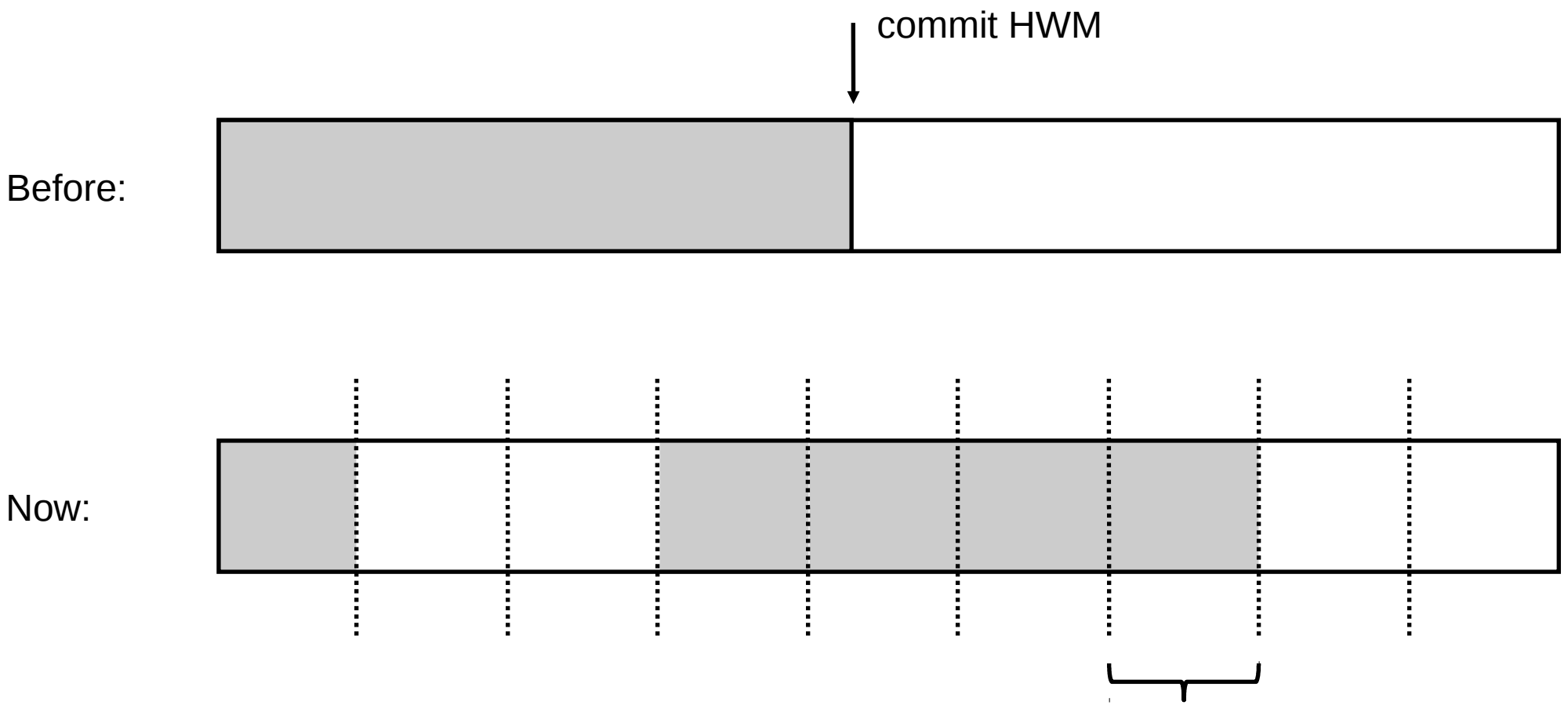
- Delay committing chunks until they are actually used
    - Loader commits as needed (like a thread stack)
    - Removes the penalty of handing out large chunks to class loaders
    - Mitigates intrachunk waste problem (at least for large chunks)

# Concern: keep number of virtual memory areas low

- (Linux): we decommit with mmap(MAP_NORESERVE) && mprotect(PROT_NONE)
  - Higher commit/uncommit fragmentation results in higher number of VMAs
  - Kernel keeps vma structures in list and rb tree
  - Too many of them may affect vma lookup
  - And we may hit process limits

- So: keep an eye on commit granularity

# Commit granules



# Current chunk allocation scheme unsuited for uncommitting chunks

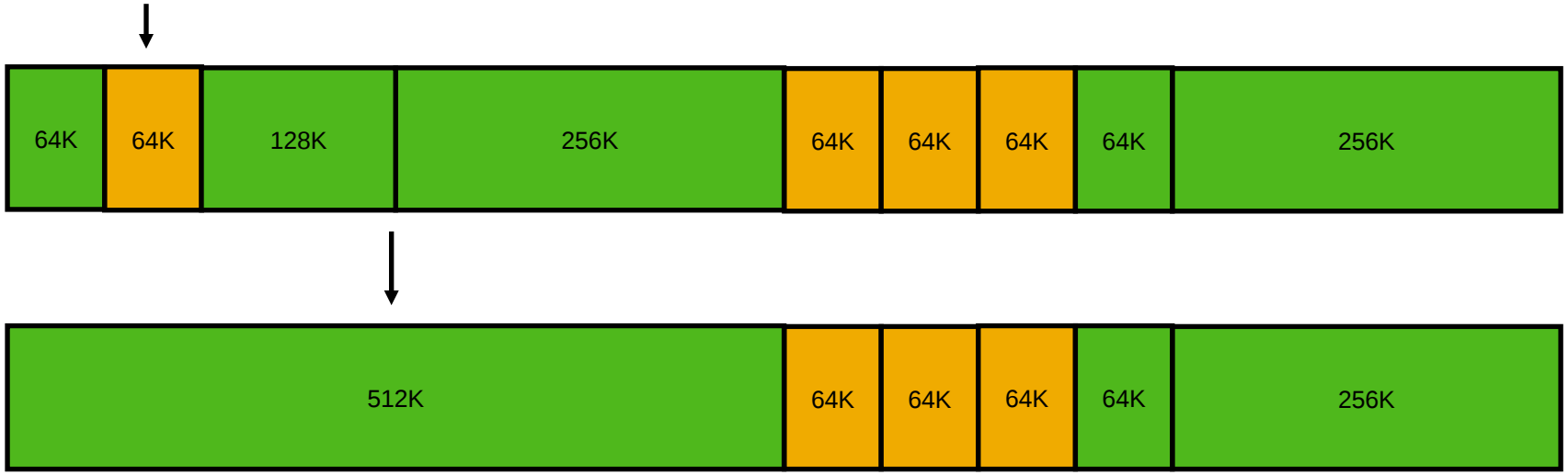
- Odd chunk geometry
  - Difficult to merge and split
  - High fragmentation
  - Complex code

- Chunk headers are a problem

# 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

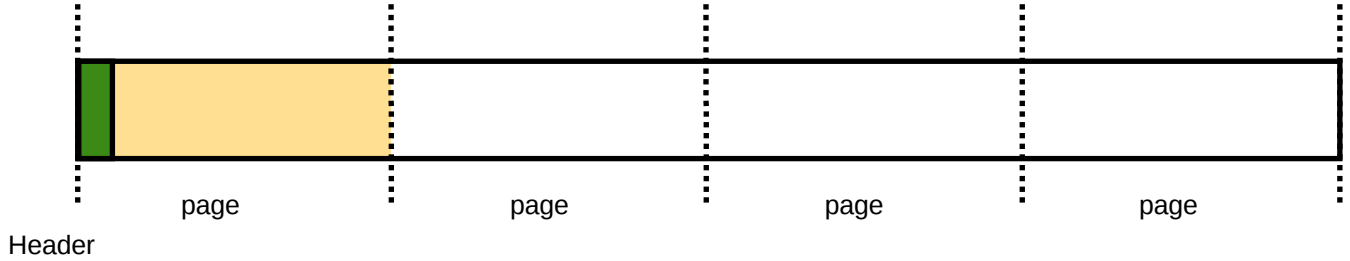
# Buddy allocator: Deallocation



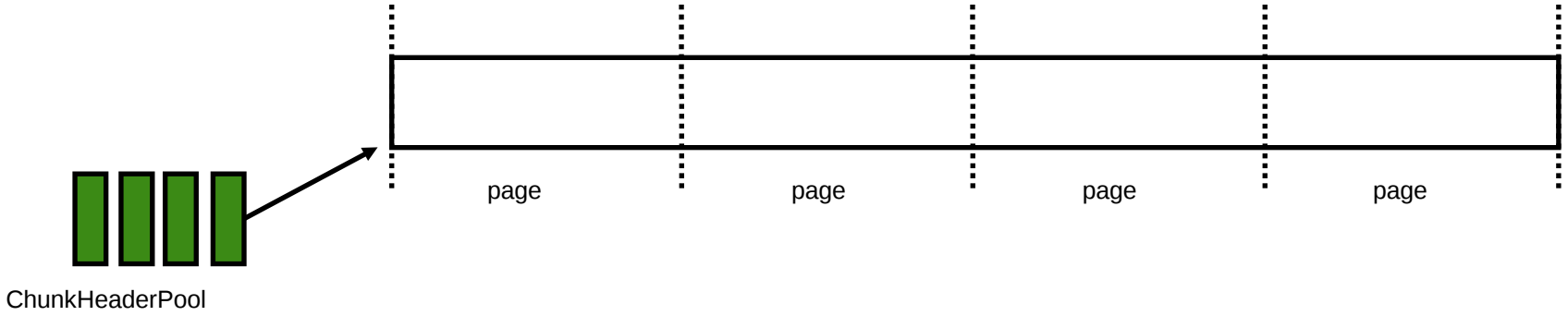

- Mark chunk as free
- If buddy chunk is free and unsplit: remove from freelist and merge with chunk
  – Repeat until root chunk sized reached or until buddy is not free
- Return result chunk to free list

# Chunk headers needed to go

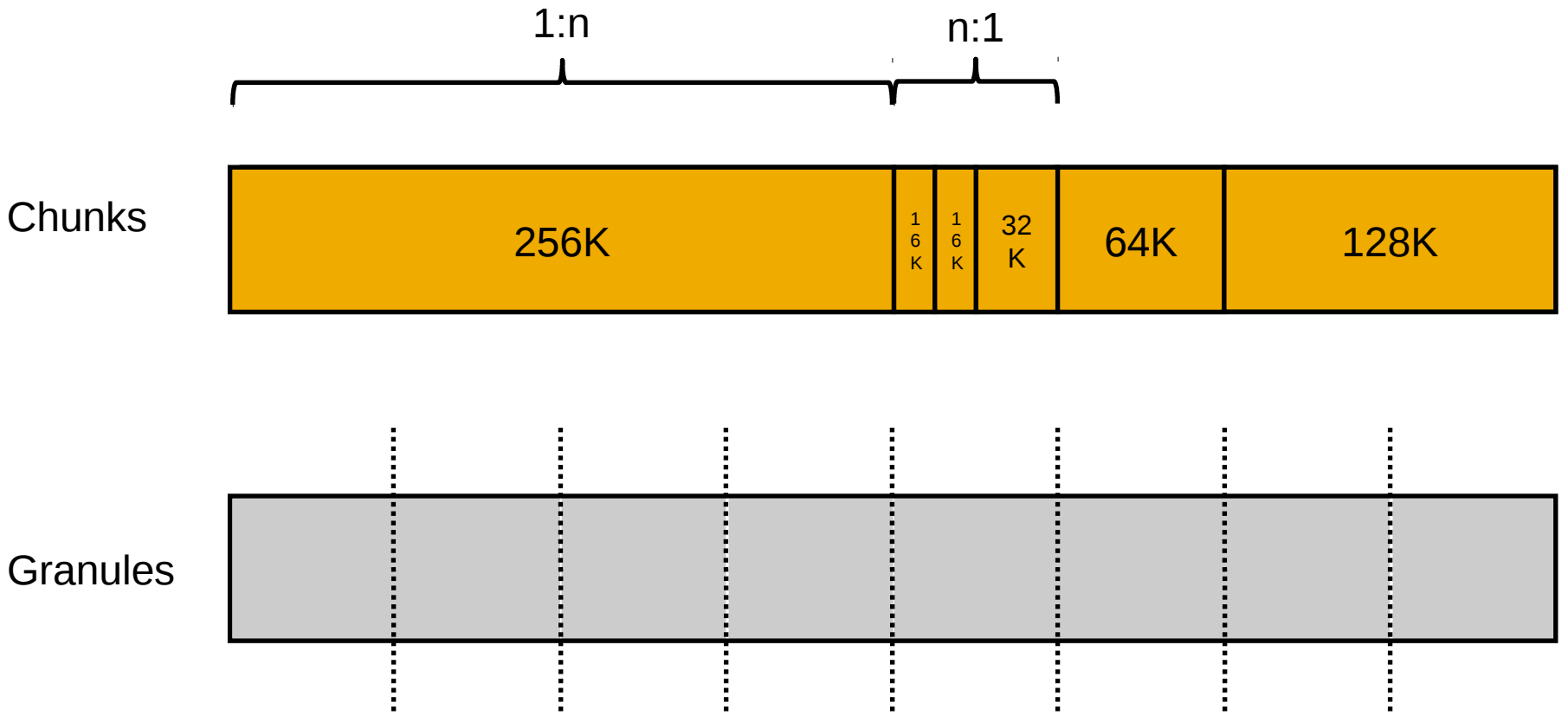
Before




Now

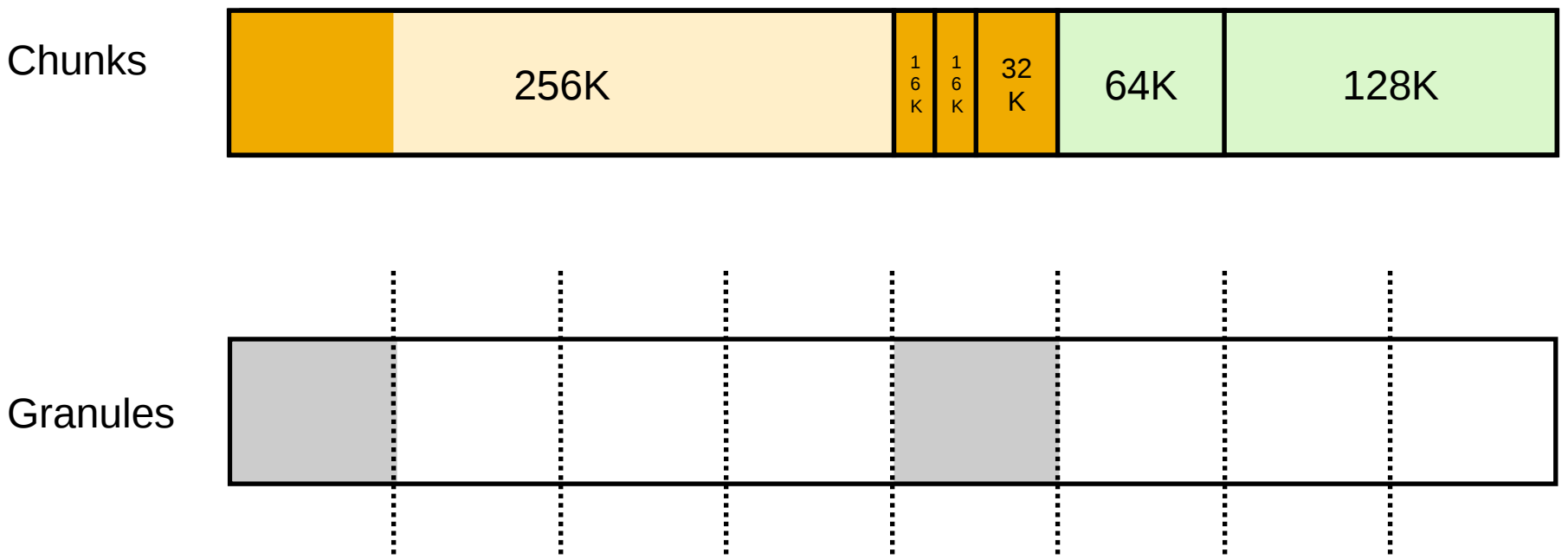


# Granules and chunks




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

# Granules and chunks

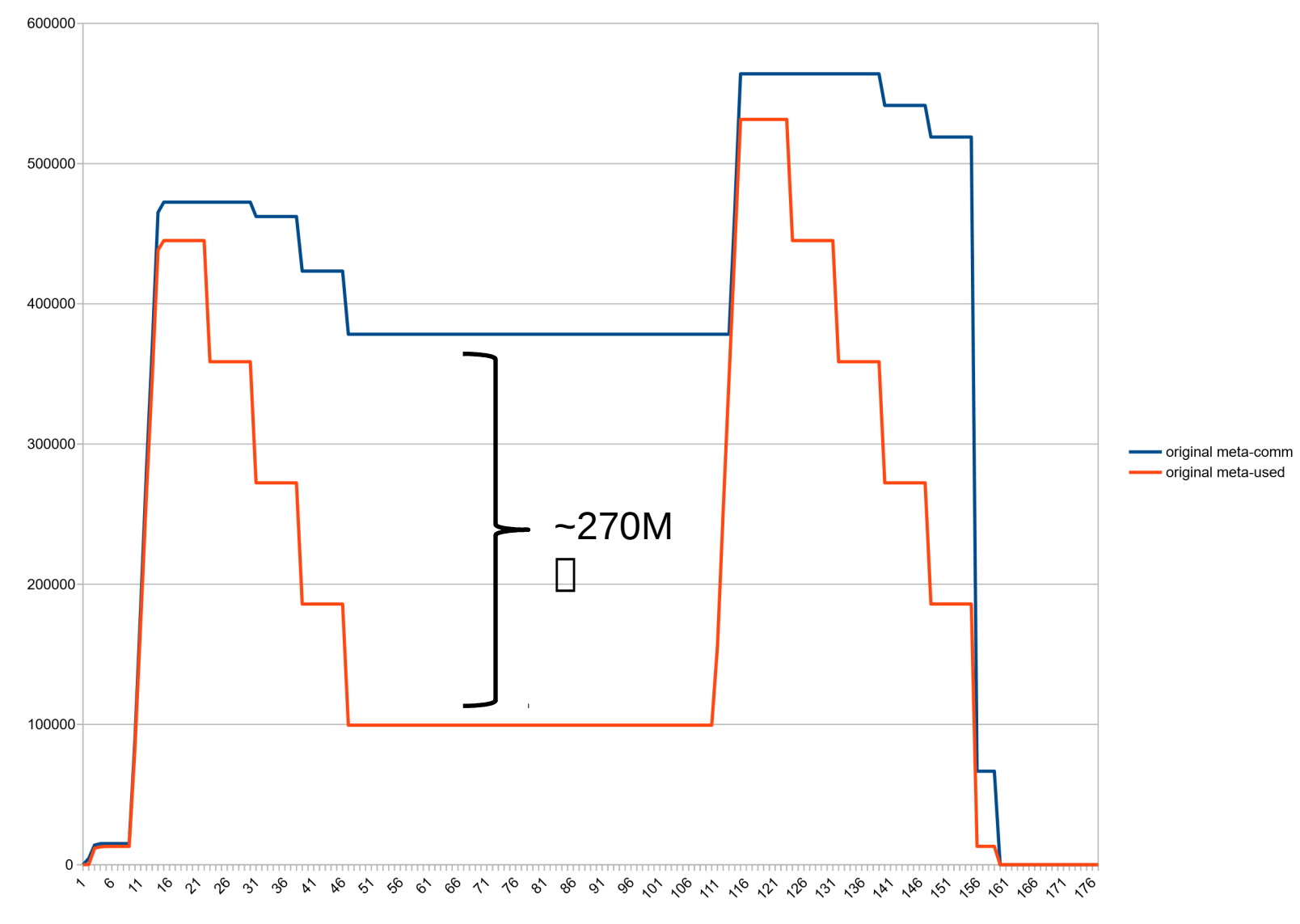



- Free chunks spanning 1+ granules can be uncommitted
- A chunk spanning >1 granules can be committed on demand

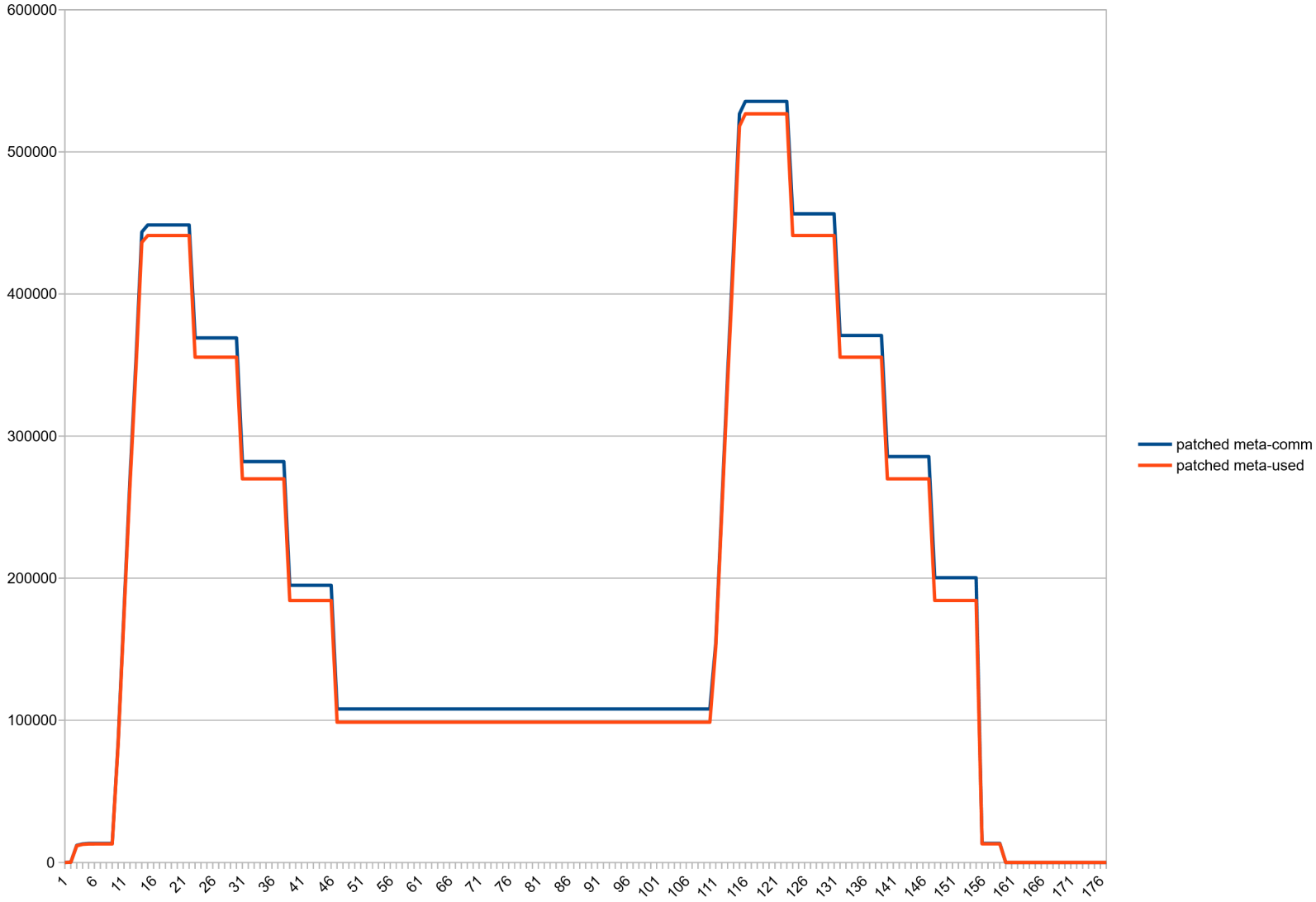
# What else changed

- Got rid of humongous chunks :)
- Got rid of occupancy map
- Better deallocation 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.

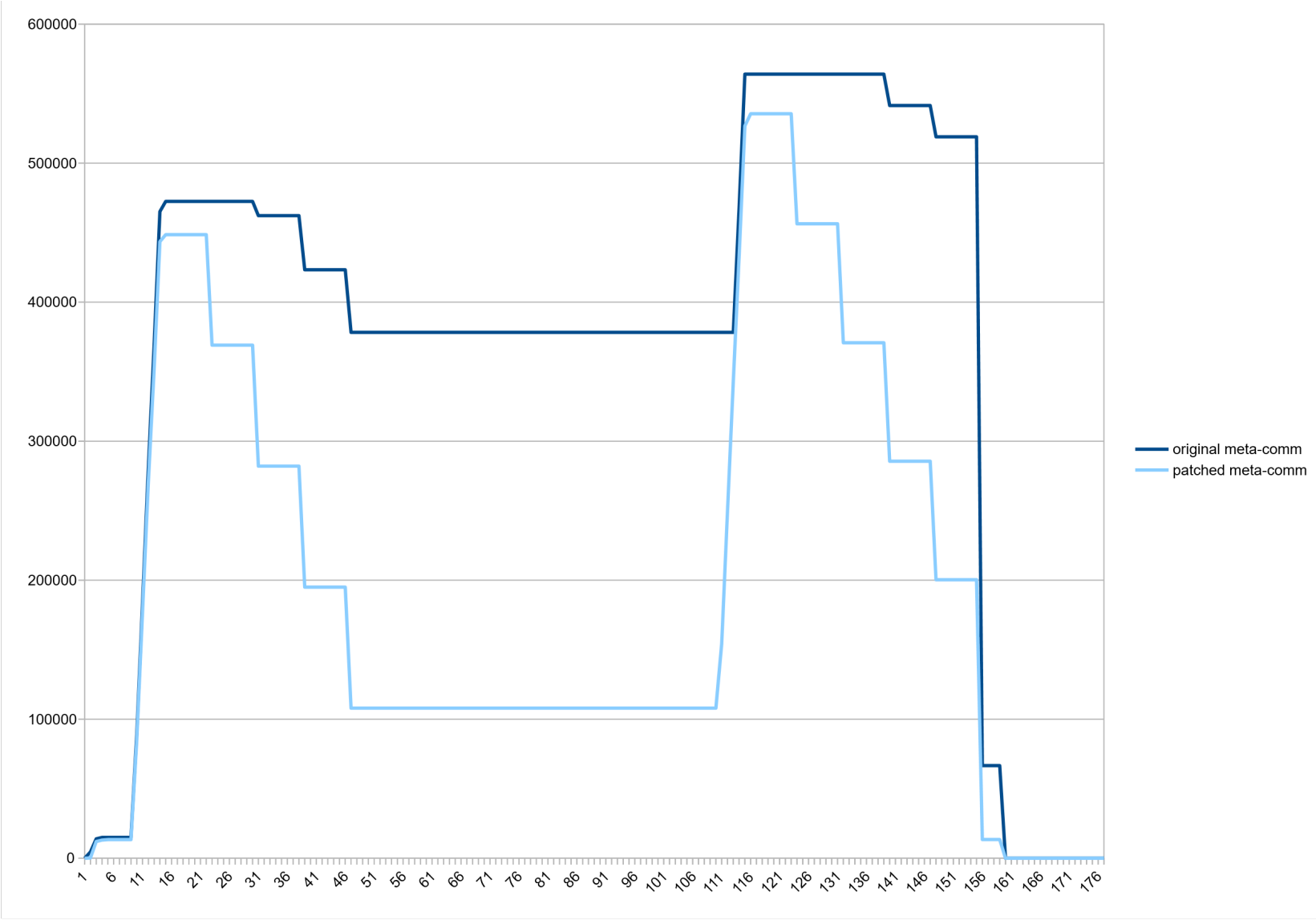
# Result: Committed vs used, Stock JDK14



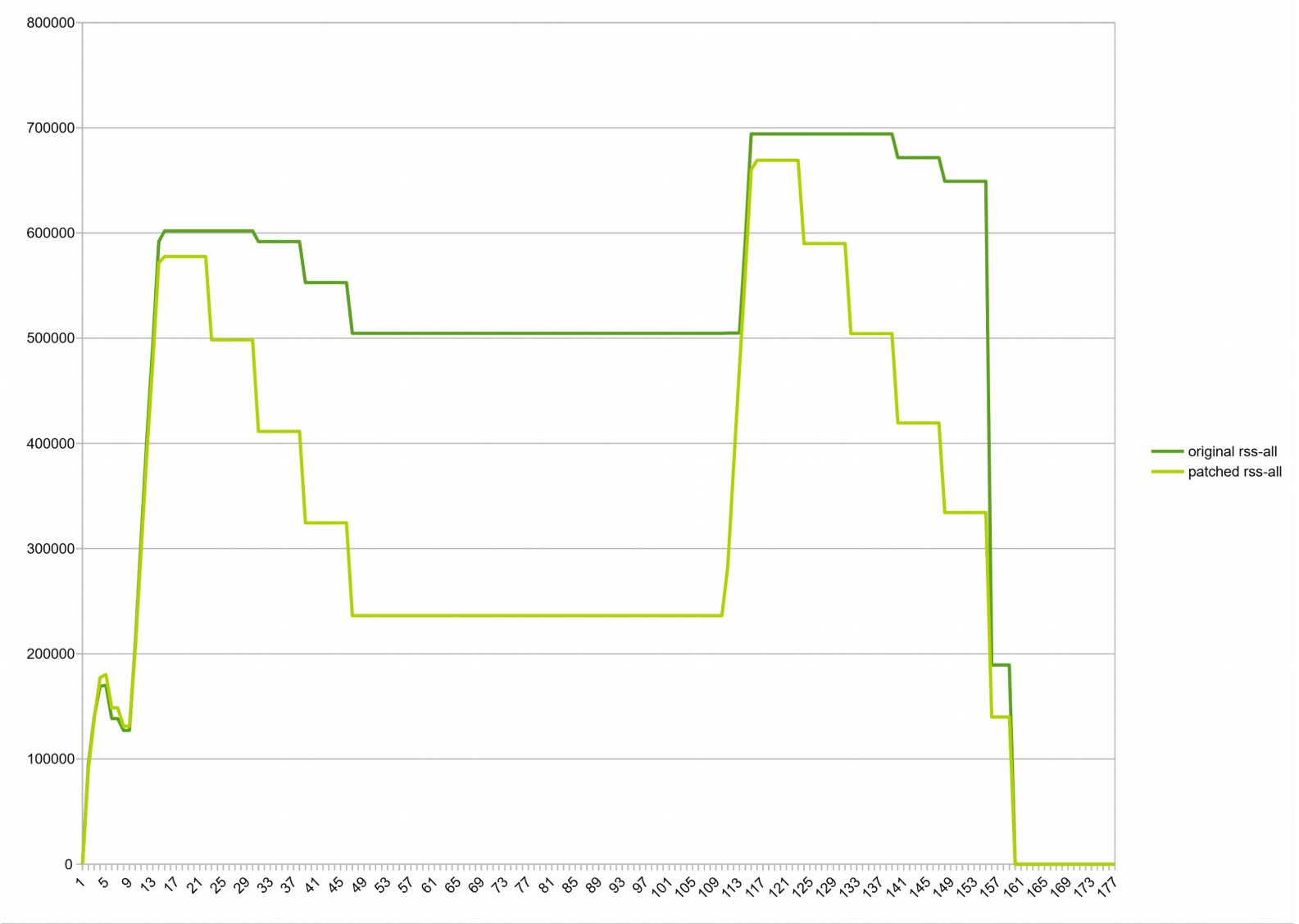
# Result: Committed vs used, Patched JDK14



# Result: committed Metaspace, Stock vs Patched VM



# Result: RSS, Stock vs Patched VM



# Bottomline: Results

- Metaspace is now elastic. Freelist problem solved (almost).

- Modest decrease in consumption even without mass class unloading
    - Wildfly standalone after startup: 61m->54m, -7m, (11%)
    - Eclipse CDT, hotspot project after C++ indexing: 138m->129m, -9m (12%)
    - jruby helloworld.rb (invokedynamic, compile=FORCE): 41m->38m, -3m, (1.2%)
    - → But, still more ideas and room for improvement.

- Better overall code quality
    - Better separation of concerns, better testability, more generic coding

# How do we go from here?

- Patch is stable. Needs more tests and may need smaller fixes but it works.
- Patch lives in jdk/sandbox repository, branch "stuefe-new-metaspace-branch“
  - http://hg.openjdk.java.net/jdk/sandbox/
- JEP exists in Draft state (“Elastic Metaspace”: https://openjdk.java.net/jeps/8221173 )
- JDK15?
  - Very difficult to bring such a large patch upstream
- 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