Elastic Metaspace (Upstream Sales Pitch)

Thomas Stüfe, SAP
March 2020

PUBLIC

THE BEST RUN

Agenda

- Motivation
- Basics
- Current implementation
- New implementation proposal

Motivation

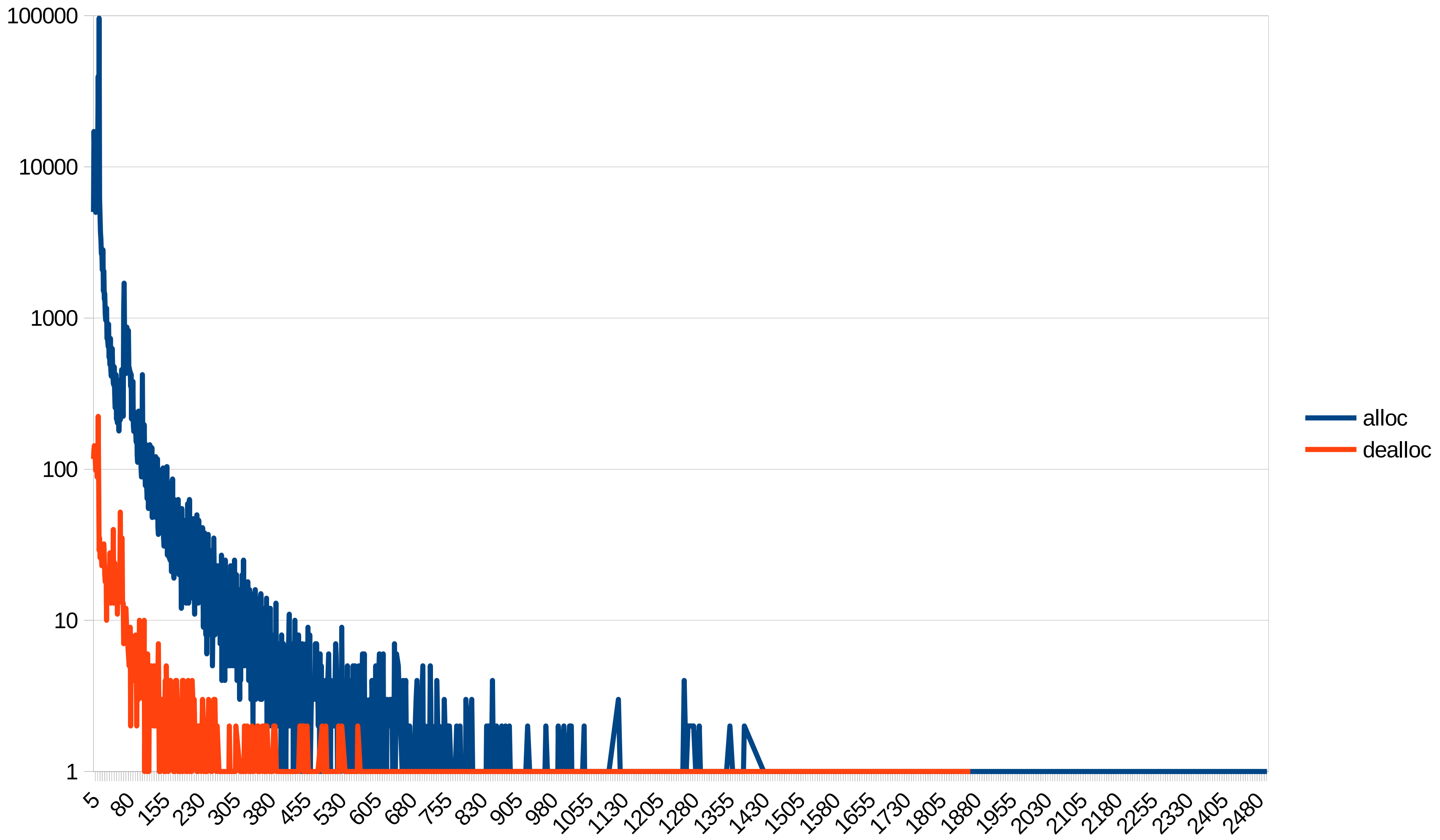
- Reduce Memory Footprint
- Keep Metaspace coding maintainable

Basics

Metadata lifecycle

- Metadata are typically allocated when classes are loaded
- Accumulated metadata is freed after class loader has been collected
  - Bulk free scenario -> No need to track individual allocations -> arena style allocation
- Exceptions: premature deallocation possible but atypical

Metadata allocation / deallocation histogram (blocks/word size)



- Taken during a wildfly startup (standalone, no apps running)

- Small block allocations dominate. Only about 1% of allocations larger than 40 words.

- Deallocation: about 1:1000, similar curve.

Why bother?

Why not use a general purpose allocator (malloc, dlmalloc, boost etc?)

We think we can do better:

- Arena style allocation is fast and memory efficient.
- We know the size distribution of typical allocations

- malloc in particular would not work anyway:
  - CompressedClassSpace
  - Platform specific limitations (e.g. sbrk hits java heap)

Current implementation

Metaspace, very simplified, on one slide

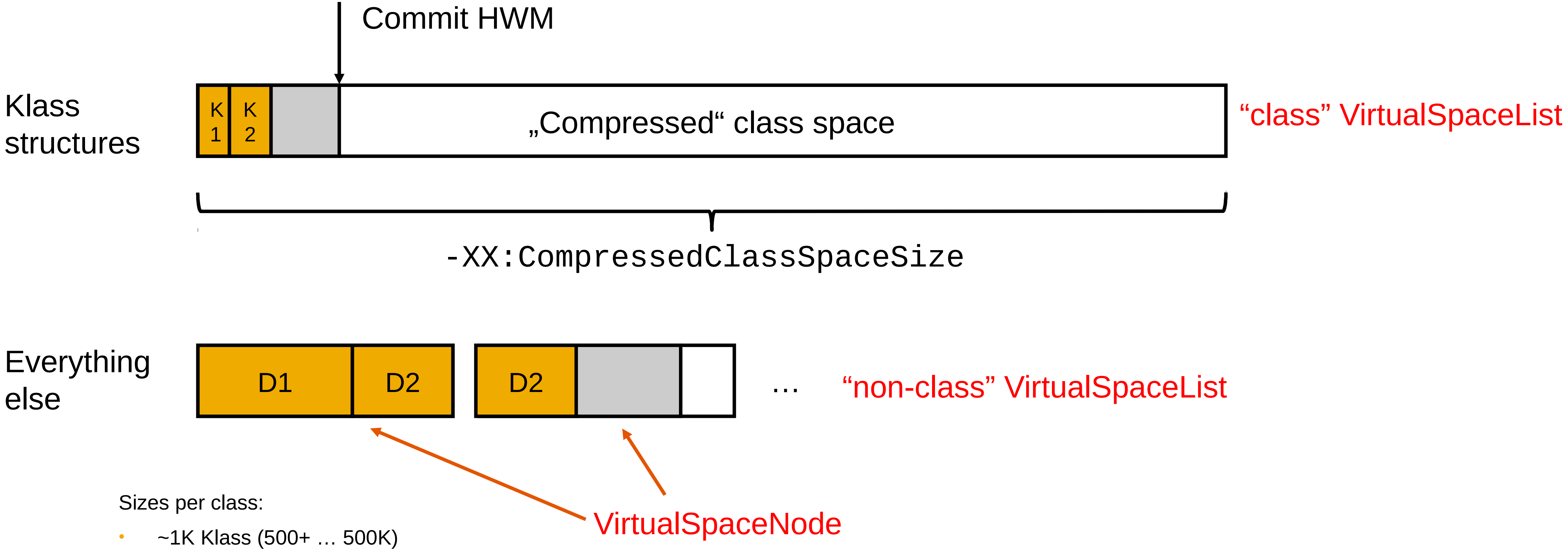
- Lowest level: a series of memory regions, mmap‘d, grows on demand, committed on demand.

- Class loaders allocate largish chunks from those regions.

- Metadata memory is allocated from those chunks via pointer bump.

- When a loader dies, its chunks are returned to a global freelist and may be reused.

Lowest level: VirtualSpaceList and VirtualSpaceNode

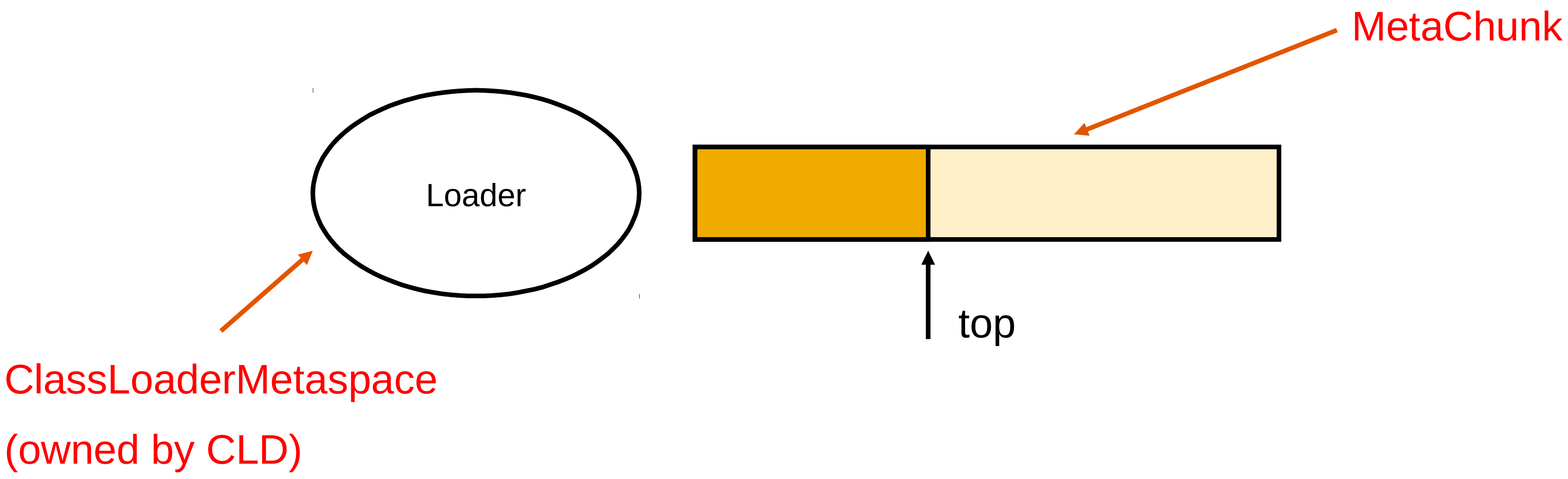


Sizes per class:

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

Current implementation

(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.

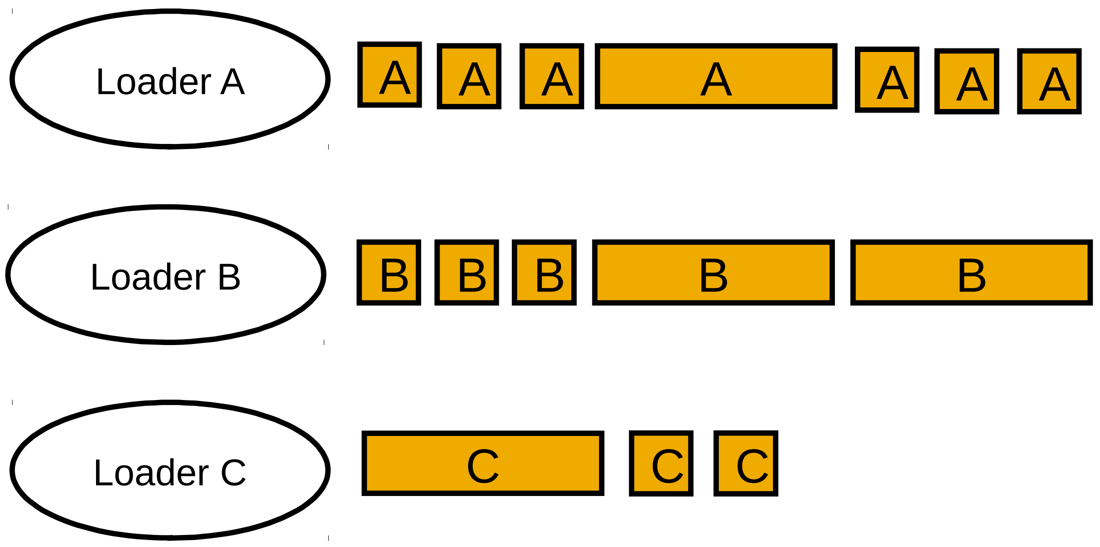
Current implementation (2)

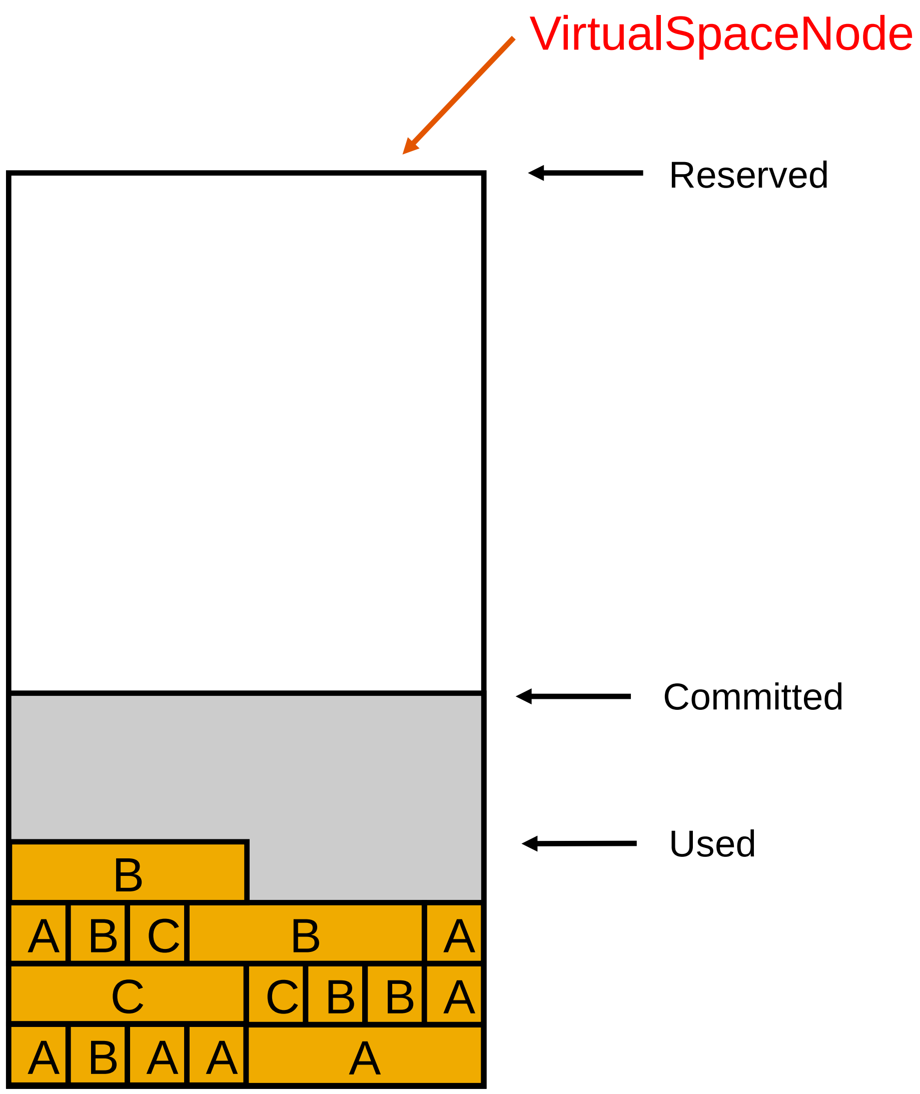
(much simplified)

- If chunk is used up, Loader acquires 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)



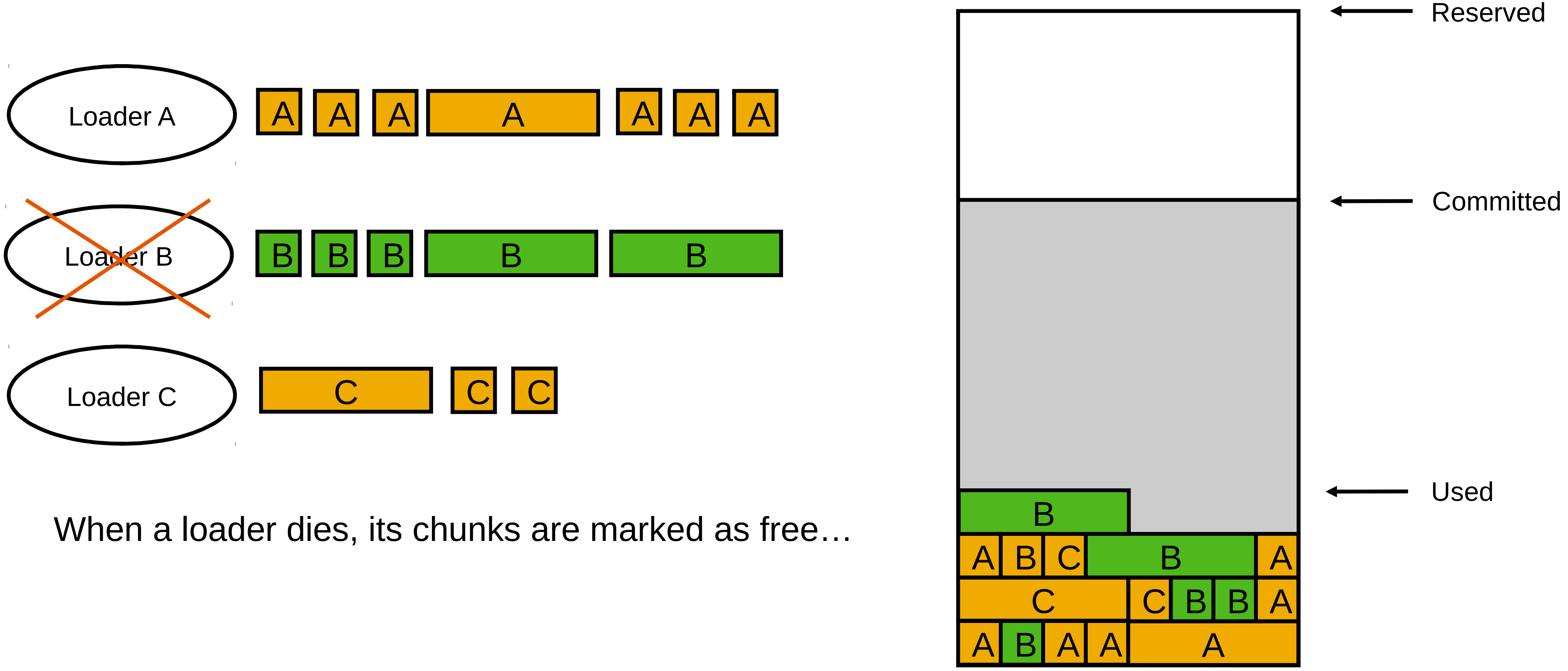


Chunks are carved from VirtualSpaceNode as they are allocated.

VirtualSpaceNode is committed on demand, never gets uncommitted.

Current implementation (4)

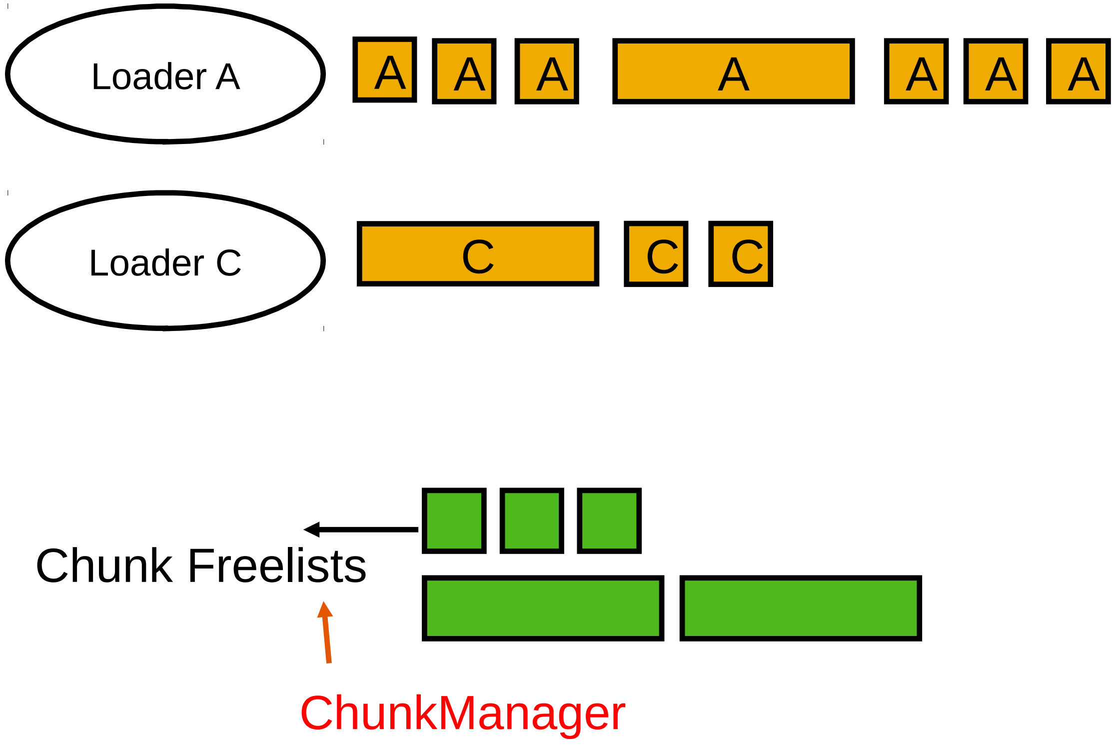
(much simplified)

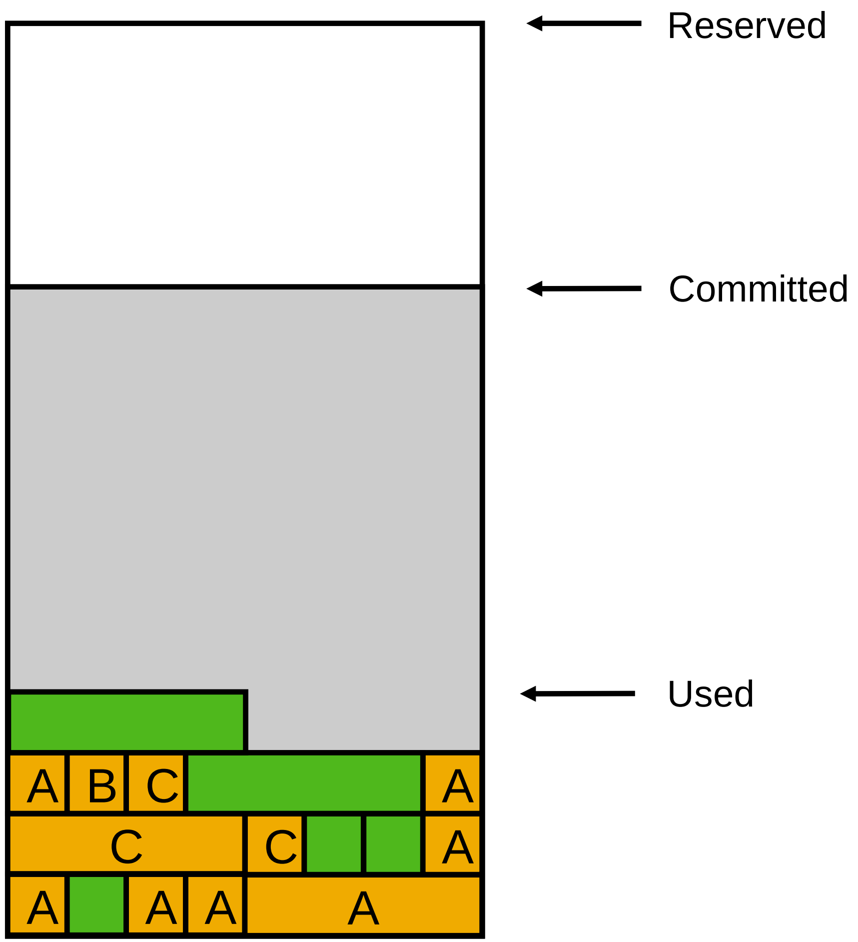


When a loader dies, its chunks are marked as free…

Current implementation (5)

(very much simplified)





…and added to global freelists (ChunkManager), sorted by size. VirtualSpaceNode is potentially unmapped.

Deallocations

- Premature release of metadata
  - 1 Class redefinitions, Profile Counters, partly loaded metadata on Class loading errors ..
  - 2 But also: remains of retired chunks (usually larger blocks)

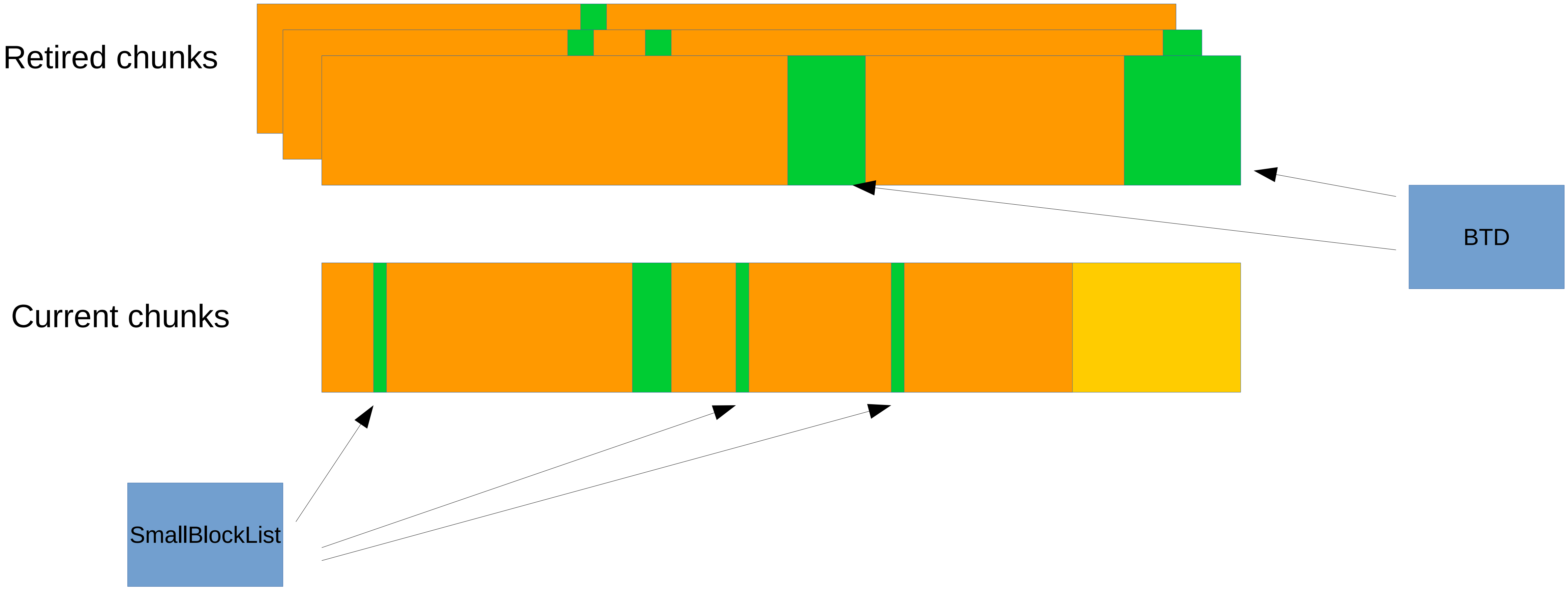
- Typically very rare (~ 1:1000 dealloc:alloc). But may happen more often in pathological cases
  - Instrumentation?

- Deallocated metadata are still owned by class loader

- We attempt to reuse them for follow up metadata allocation, to varying degrees of success

- Deallocation histogram: similar to allocation but higher number of larger chunks due to (2)

Deallocations

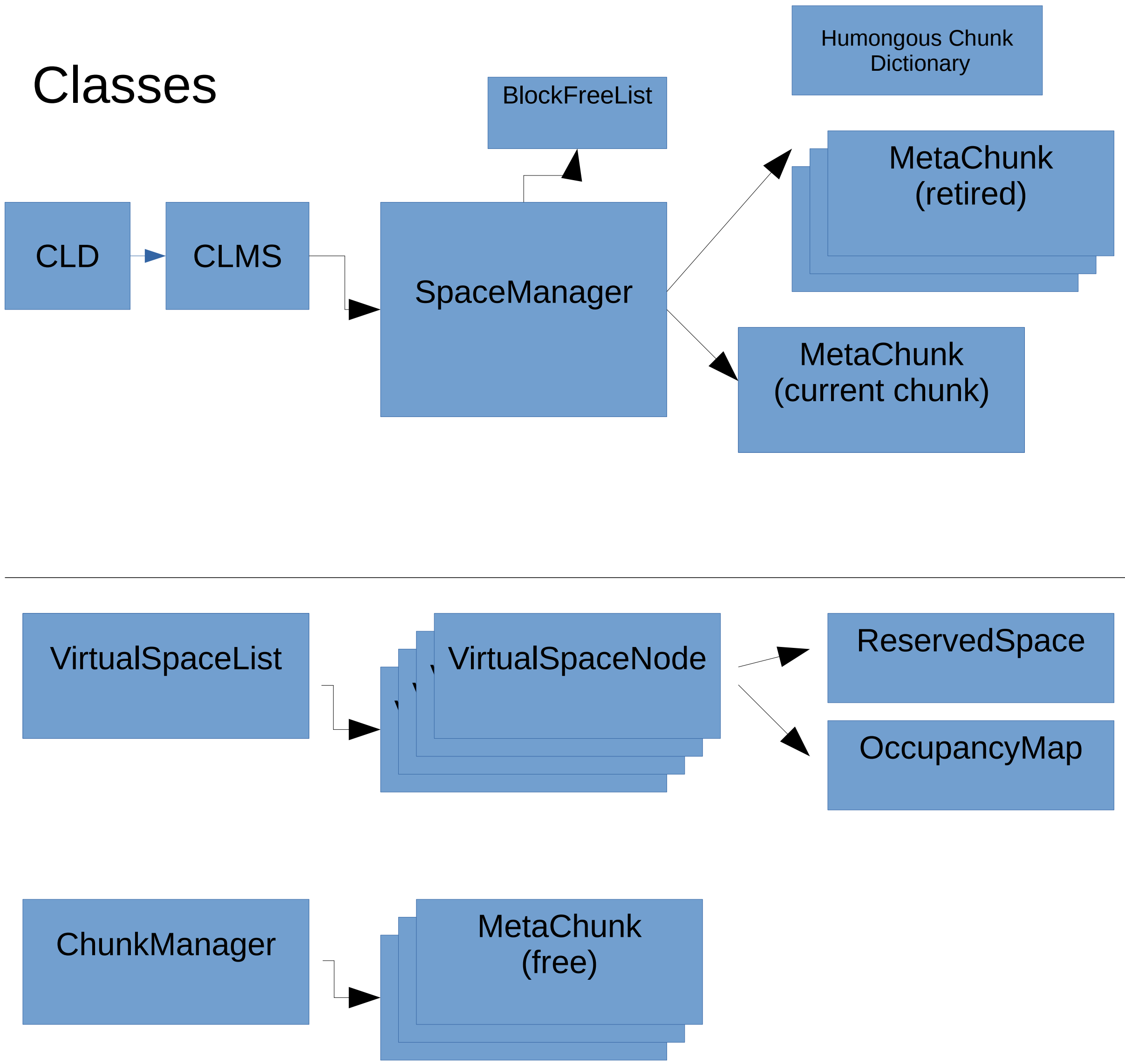


Deallocations: BlockFreeList

- Two parts
  - „SmallBlocks“ - a ordered vector of linked lists of free metadata, for small sizes (up to 13 blocks)
    - O(1) insert/retrieval
  - „BlockTreeDictionary“ - a BST for larger blocks
    - BinaryTreeDictionary
    - Unbalanced

- Whats not optimal:
  - Smallblocks does not search for larger blocks
  - Dictionary: too large blocks are retrieved and inserted back unnecessarily
  - We are hesitant to query the BTD
  - Complicated code

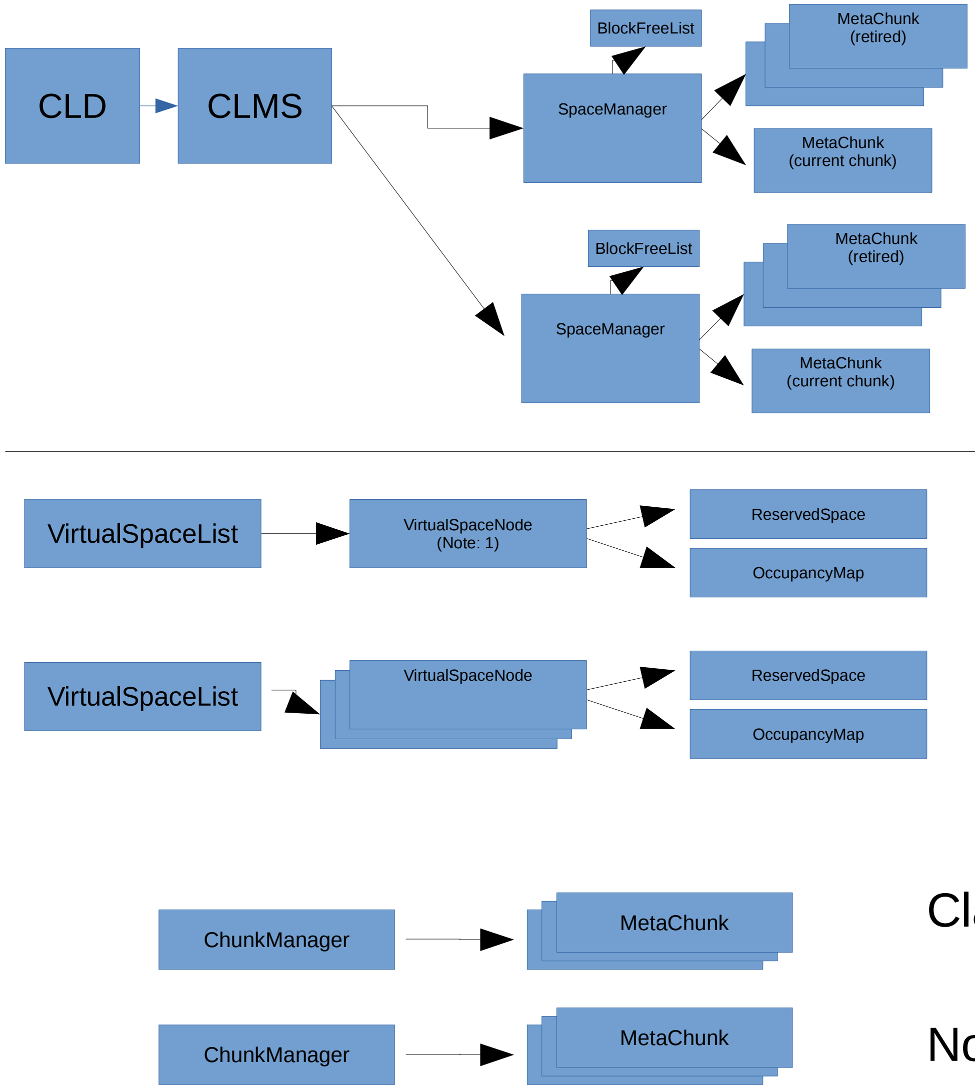
Classes



Per class loader

Global

..with CompressedClassPointers



Class metadata

“non-class” metadata

Per class loader

Class VSlist (note: really just one element).

Non-Class VSlist

Global

Class ChunkManager

Non-Class ChunkManager

Chunk sizes

- Class space: 1K (“special”), 2K (“small”), 32K(“medium”)
- Non-class space: 1K, 4K, 64K
- And humongous chunks: larger than medium and variable sized

Metachunk coalescation

- The „chunk size choking problem“
- JDK-8198423
- Since then chunks can be merged and split, within limits
  - 4x1K chunks -> 4K
  - 16x4K chunks -> 64K
- Basically the whole thing is now a weird buddy allocator
  - A bit inefficient due to the odd chunk geometry
  - But it solved the problem
  - But I was afraid to touch too much, so the whole patch is one gigantic band aid
  - Ugly and difficult to maintain :-(

Concern: „Micro Loaders“

- Some loaders / CLDs only ever load one class:
  - Anonymous classes
  - Reflection delegators

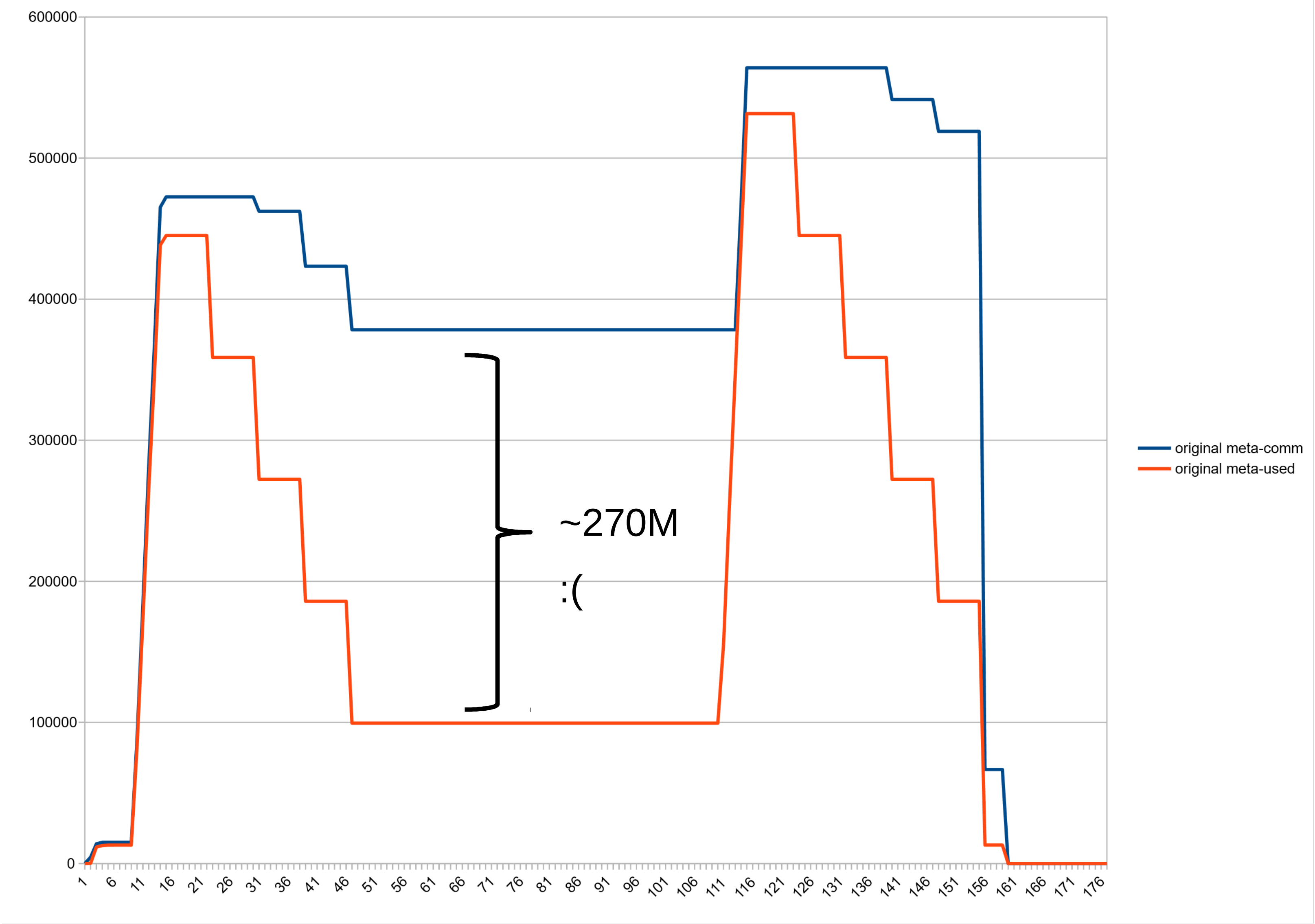
- Not optimal.
  - In class space, only one InstanceClass is allocated, but needs a whole chunk
  - In non-class space, ~10-20 allocations

Main waste areas

- 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 (and deallocated space waiting for reuse) is wasted
  - Worse with micro loaders, if there are a lot

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%)
```

Eat up! - Intrachunk waste from idle loaders

- Most loaders stop loading at some point. Remainder space in current chunk as well as deallocated blocks remain unused, effectively wasted.

- How large a chunk do we give to a loader?
    - Too small: high fragmentation and contention on central allocator parts
    - Too large: wasted space
    - → we try to guess future loading behavior. We may guess wrong and suffer.
- We have tuned these areas a lot (see e.g. Zhengyu Yu's work on JDK-8190729 and JDK-8191924) but the solutions are far from optimal

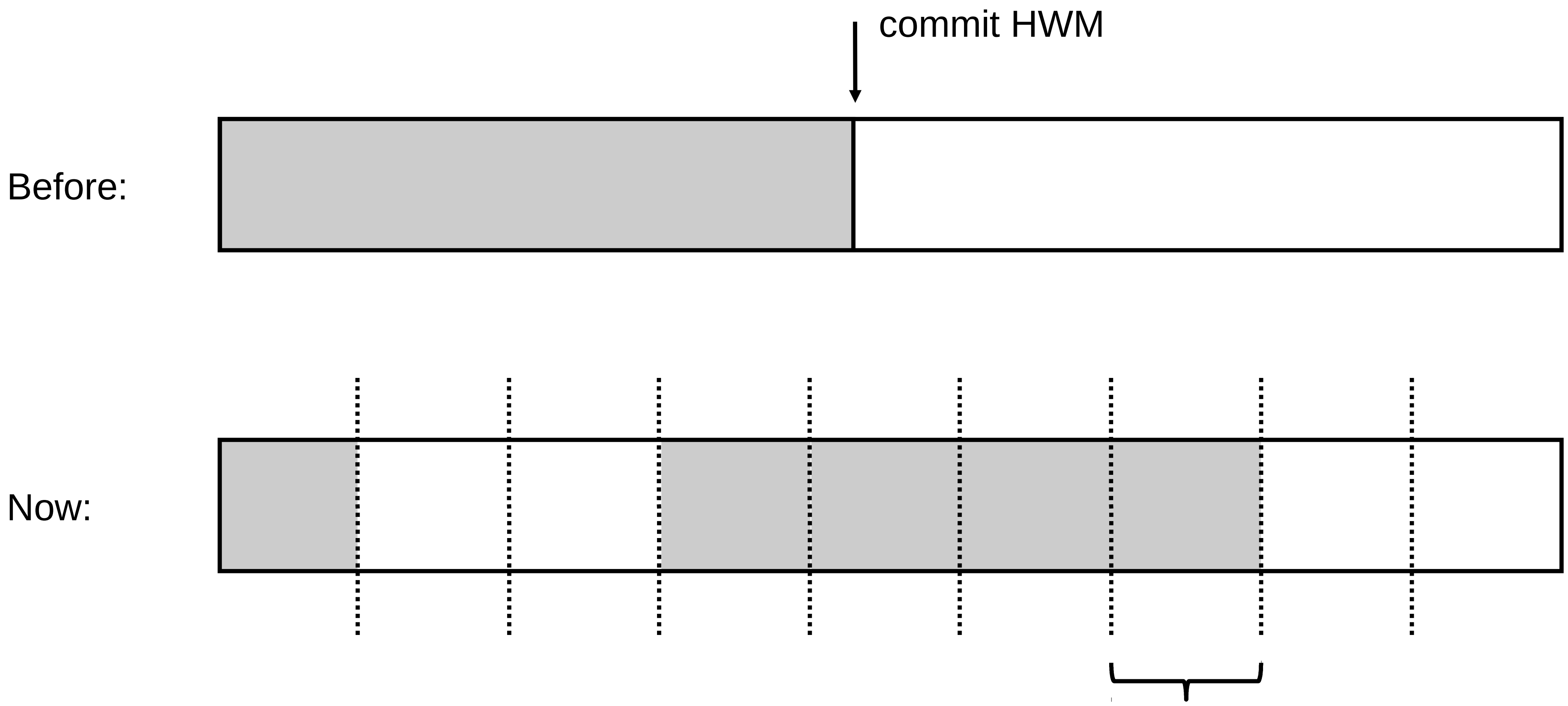
Reimplementation

Basic idea

- Chunks in freelists can be uncommitted

- 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

Commit granules



Granule size

Adjustable, defaults to 64k

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

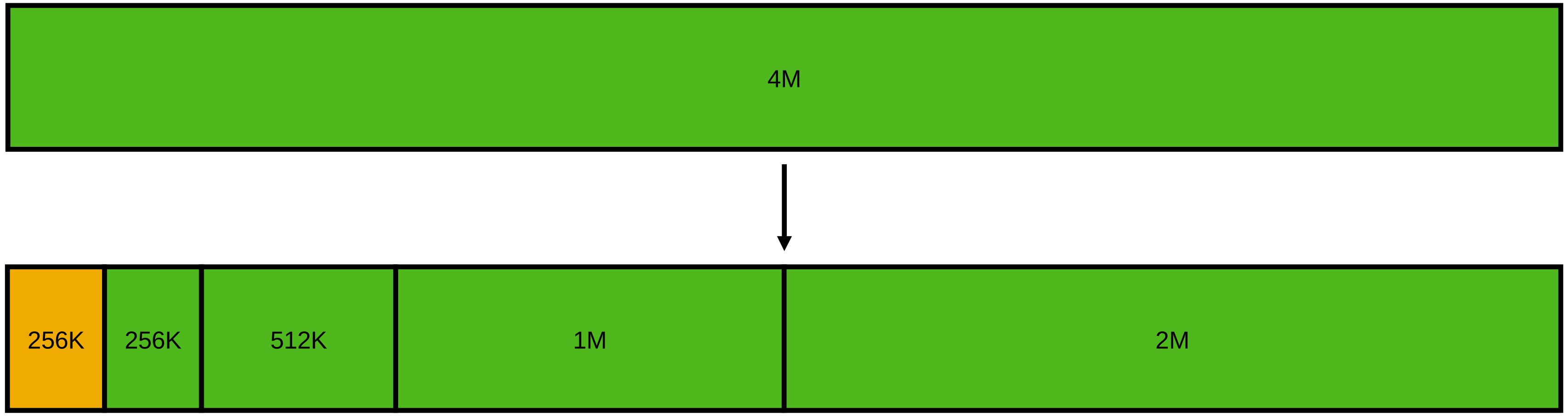
Pow 2 based buddy allocator for chunks

- We need a better, cleaner chunk geometry scheme.
- Power 2 based buddy allocation is a nice fit.
- Chunks sized from 1K … 4M in pow2 steps (13 sizes in total).
- Dead simple to split and merge.
- Low external defragmentation -> Leads to larger free contiguous areas.
- Standard algorithm widely known

Pow 2 based buddy allocator for chunks

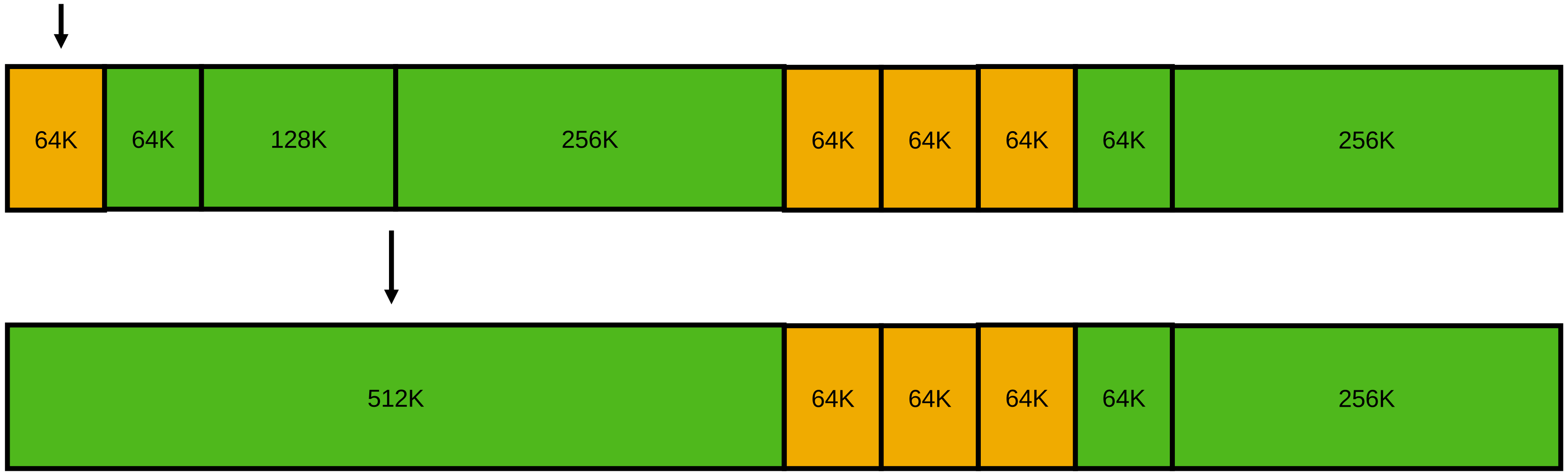
```
// Each chunk has a level; the level corresponds to its position in the tree
// and describes its size.
//
// The largest chunks are called root chunks, of 4MB in size, and have level 0.
// From there on it goes:
//
// size    level
// 4MB     0
// 2MB     1
// 1MB     2
// 512K    3
// 256K    4
// 128K    5
// 64K     6
// 32K     7
// 16K     8
// 8K      9
// 4K      10
// 2K      11
// 1K      12
```

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

New VirtualSpaceNode

- Metaspace is now segmented into “root chunk areas”: 4MB sized, aligned to 4MB

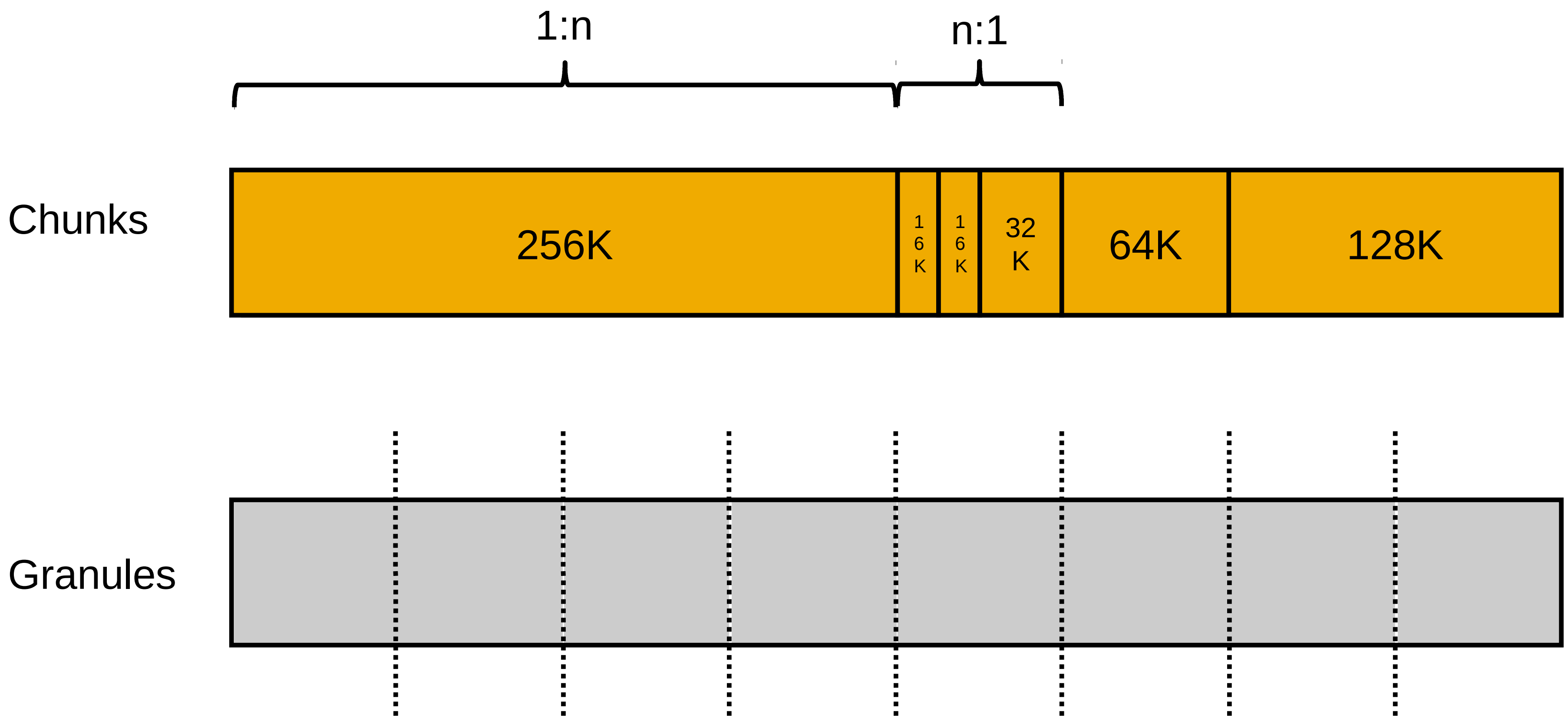
- VirtualSpaceNode:
  - now only allocates root chunks → much simplified code.
  - Does not commit! But provides committing as service to upper layers.
  - Keeps a bitmap to keep track of committed/uncommitted granules.

New Chunk Manager

- ChunkManager keeps 13 freelists, one per chunk level

- Allocation flow:
  - SpaceManager to ChunkManager: give me chunk of level X
  - ChunkManager: have one in freelist(X) – ok.
    - Have none? Search upward in freelists(X+1,2…)
    - Found a larger chunk? Split it buddy style; return chunk of level X; put splinter chunks back into respective freelists
    - Found no larger chunk? Ask underlying VirtualSpaceNode for new root chunk. Proceed as described above.

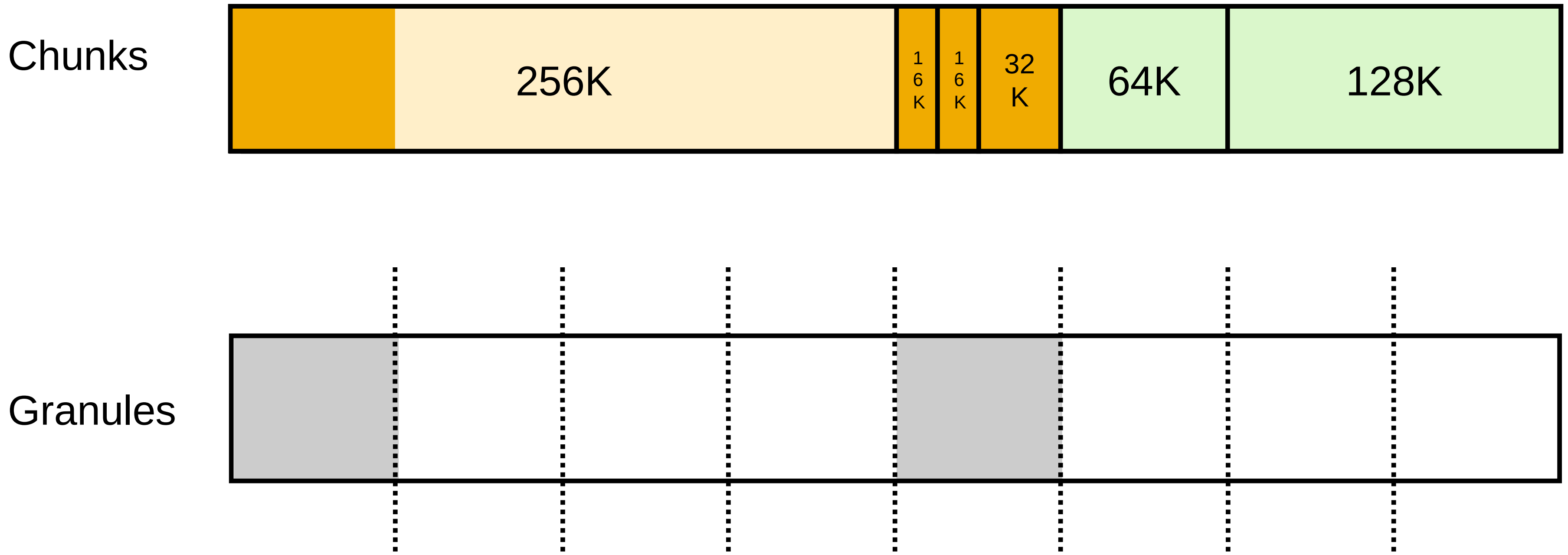
Granules and chunks – putting it all together



A larger chunk can span multiple granules (1:n)

Multiple small chunks can cover a single granule (n:1)

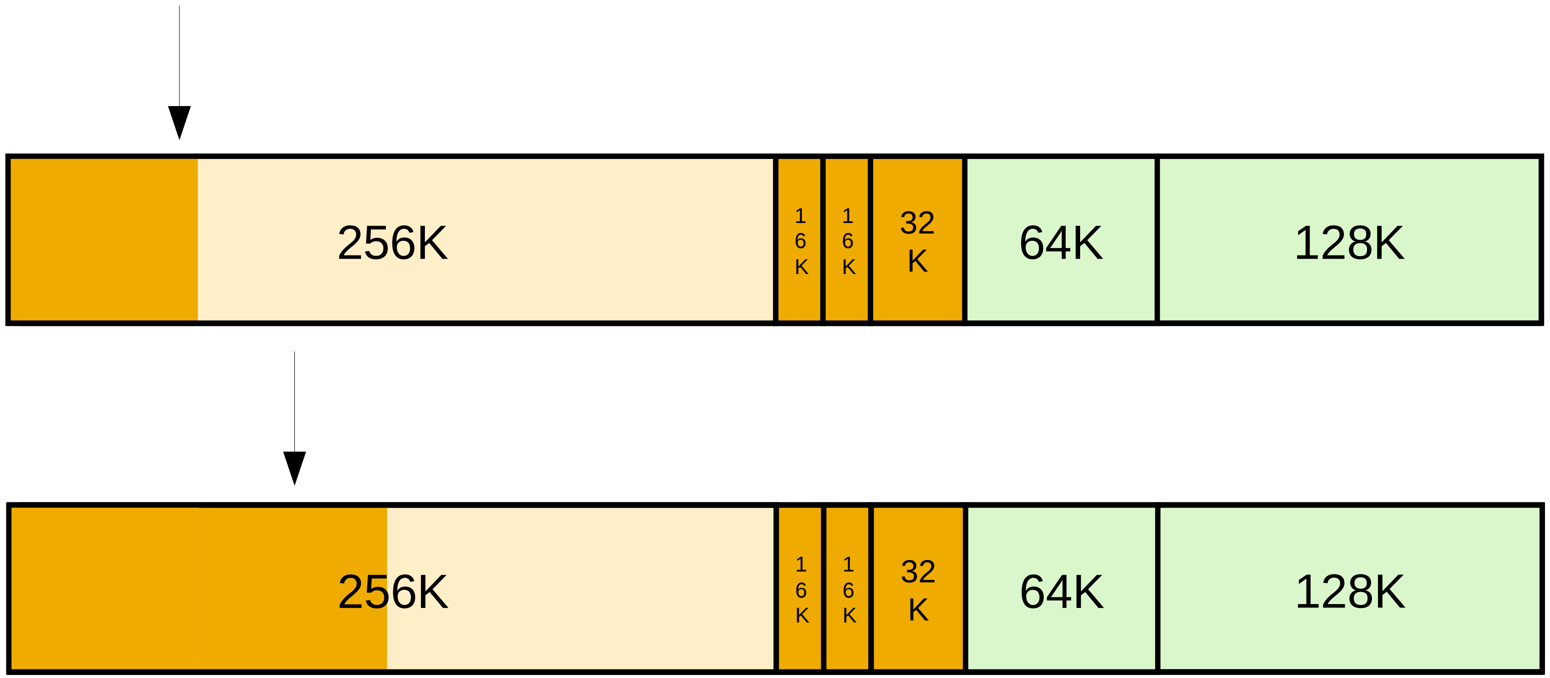
Granules and chunks – putting it together



Free chunks spanning 1+ granules can be uncommitted

A chunk spanning >1 granules can be committed in parts, on demand

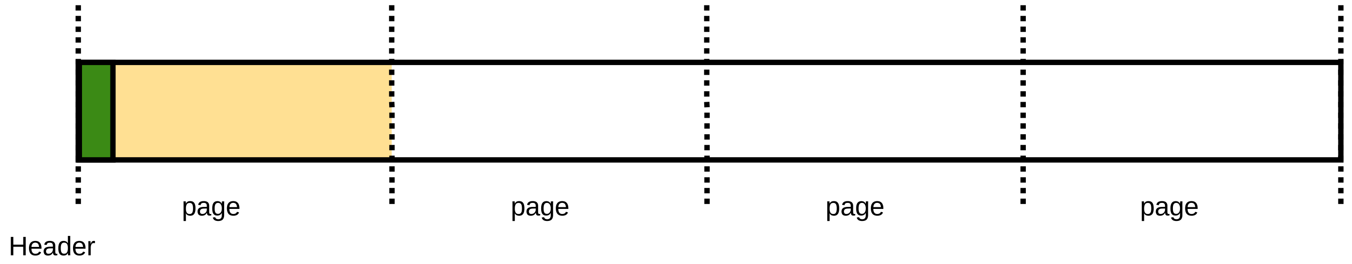
Larger chunks can be committed on demand



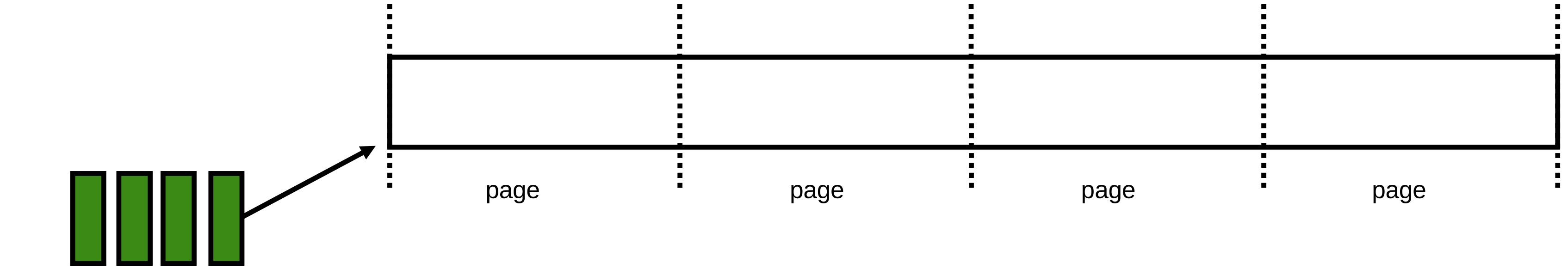
Removes penalty for guessing wrong in the “how large a chunk should I give him” guessing game at least for larger chunks (e.g. for the boot loader).

Chunk headers needed to go

Before



Now



New Deallocation handling („LeftOverManager“)

- Bin list (similar to SmallBlocks) + a newly written BST
  - New binlist covers more sizes (atm 32) and searches also upward
  - The new BST is similar to the old one, but much reduced in code size and much simpler.
  - New BST knows its largest node size.
  - Note: we do not need the binaryTreeDictionary anymore :)

- We now split blocks where it makes sense.

- Possible further improvement: make it an RB tree.

What else changed

- Got rid of humongous chunks :)
- Got rid of occupancy map
- Nice: Chunks can now grow in-place
  - Saves overhead and reduces intrachunk waste
- Code cleaner and more maintainable; better separation of concerns and testability.
- Lots of new gtests

More improvements possible

- Improve error analysis for overwrites in Metaspace
  - Simple: Buffer zones, canaries, disabling deallocation
  - Costly but possible: guard pages

- Better micro loader handling

Concern 1: keep number of virtual memory areas low

- (Linux):
  - 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

- Solution: commit granule size is adjustable. Larger granules → lower fragmentation at the cost of lower memory returns.

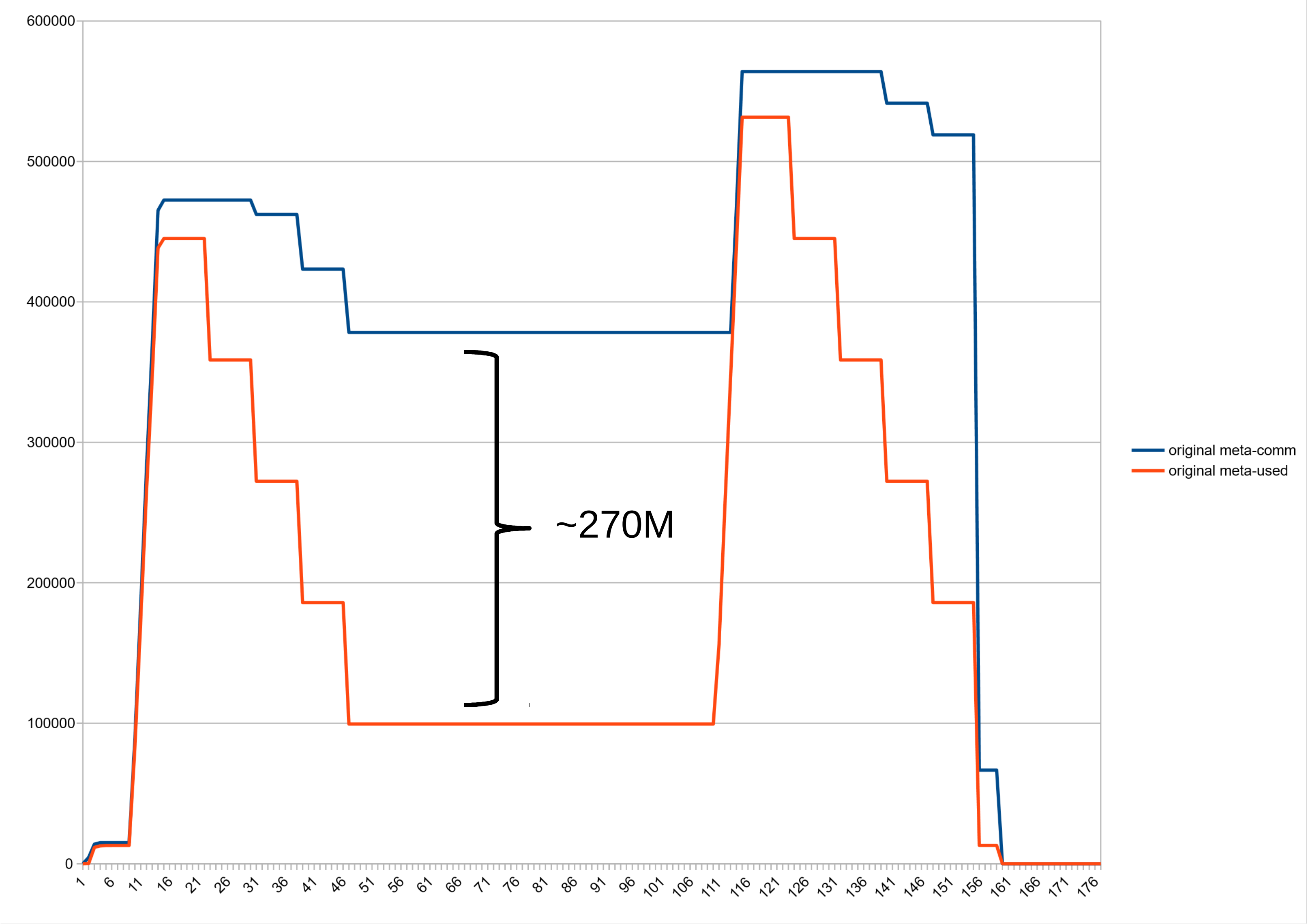
Concern 2: uncommit speed

- Matters only to GCs which have no concurrent class unloading

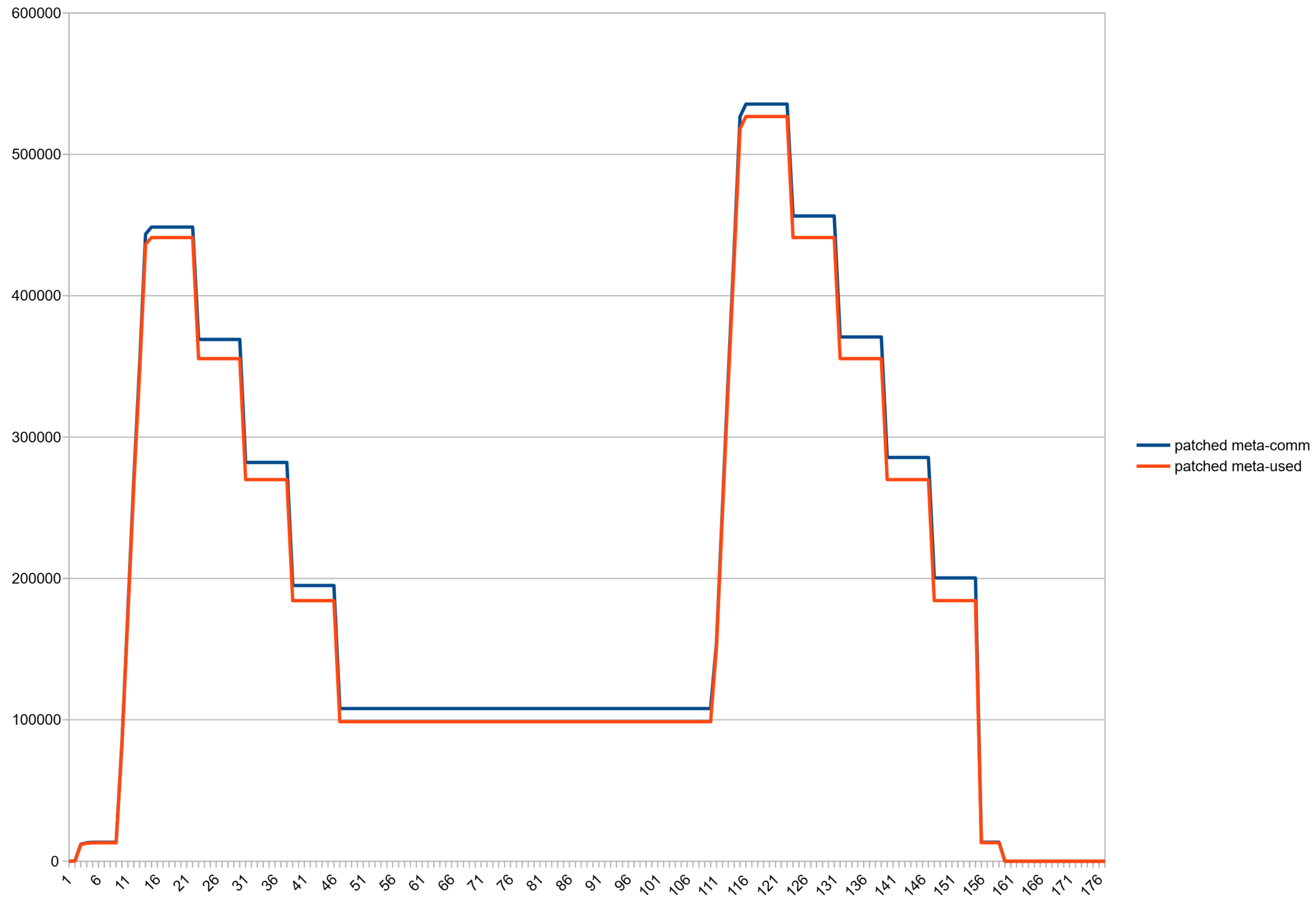
- (Linux):
  - Page table has to be unrolled & deallocated. How expensive this is depends on population of uncommitted area: how many pages had been committed before, and their size.
  - Hence indirectly on size of uncommit region
  - And also on number of vma in committed region, although in our case it should always be one.

- Currently I see no problems in practice, but a fall-back plan would be to uncommit concurrently. Not that complicated if we keep current locking scheme

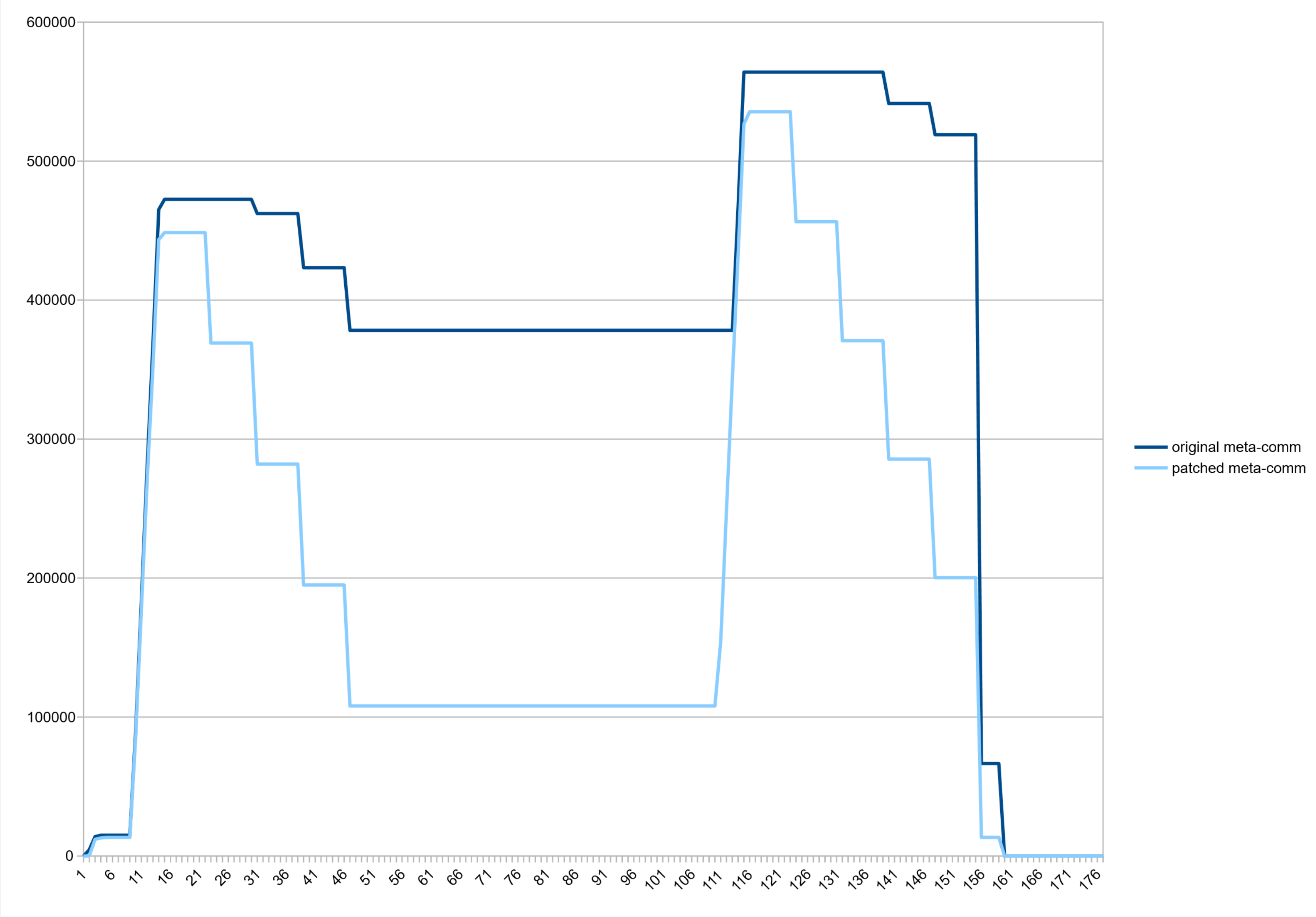
Result: Committed vs used, Stock JDK14



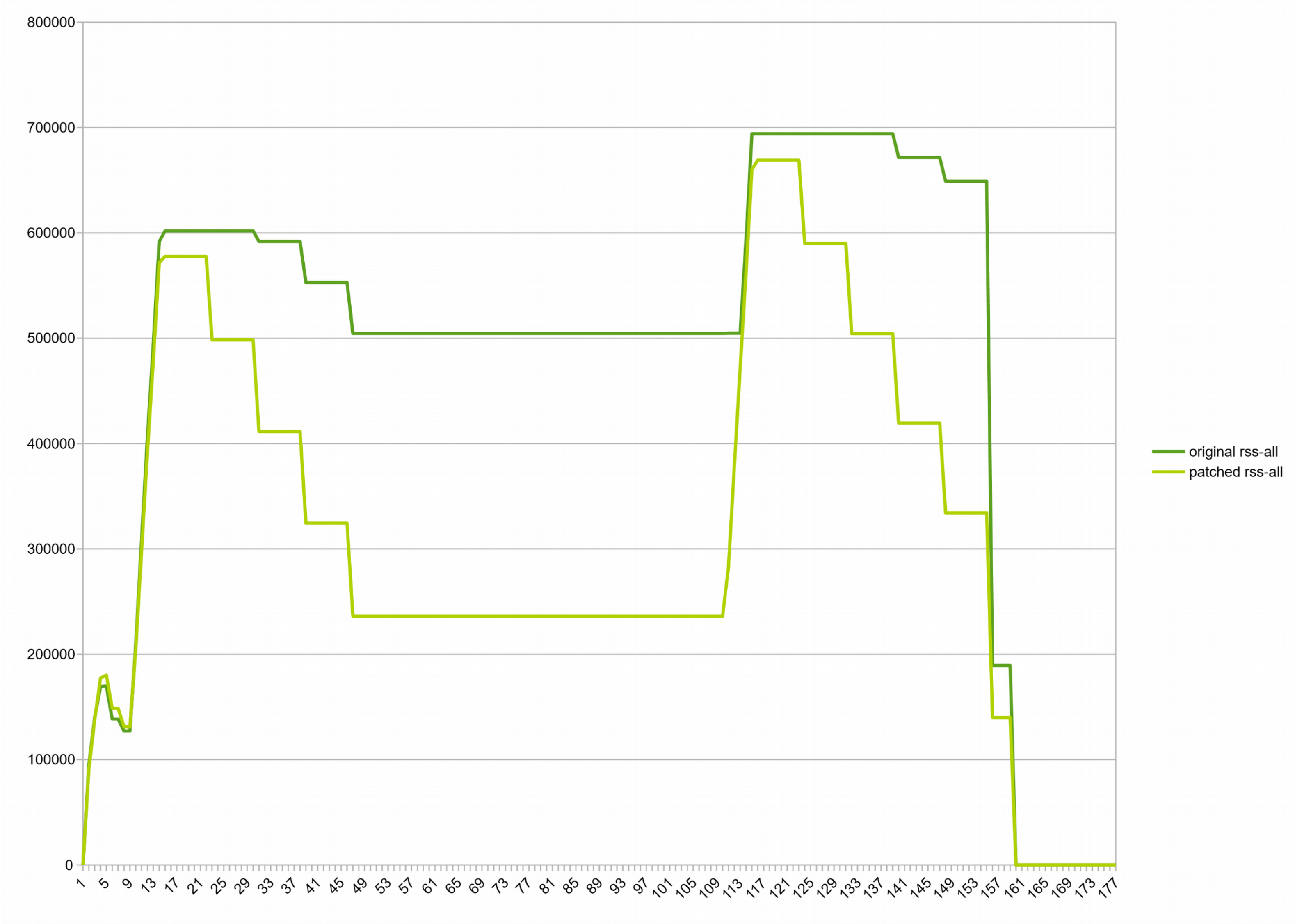
Result: Committed vs used, Patched JDK14



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, -7m, (11%)
- Eclipse CDT, hotspot project after C++ indexing: 138m->129m, -9m (12%)
- jruby helloworld.rb (invokedynamic, compile=FORCE): 41m->38m, -3m, (1.2%)

Performance costs

- jbb2015 did not show regressions
- Micro benchmarks doing mass class loading and unloading show atm 1-2% decrease in performance. I am working on it.

How do we go from here?

- Patch is stable. Needs more tests and 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?

- 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

THE BEST RUN SAP®