

Elastic Metaspaces (Reviewer Sales Pitch)

Thomas Stüfe, SAP
Feb 01, 2020

PUBLIC

Follow us



www.sap.com/contactsap

© 2019 SAP SE or an SAP affiliate company. All rights reserved.

No part of this publication may be reproduced or transmitted in any form or for any purpose without the express permission of SAP SE or an SAP affiliate company.

The information contained herein may be changed without prior notice. Some software products marketed by SAP SE and its distributors contain proprietary software components of other software vendors. National product specifications may vary.

These materials are provided by SAP SE or an SAP affiliate company for informational purposes only, without representation or warranty of any kind, and SAP or its affiliated companies shall not be liable for errors or omissions with respect to the materials. The only warranties for SAP or SAP affiliate company products and services are those that are set forth in the express warranty statements accompanying such products and services, if any. Nothing herein should be construed as constituting an additional warranty.

In particular, SAP SE or its affiliated companies have no obligation to pursue any course of business outlined in this document or any related presentation, or to develop or release any functionality mentioned therein. This document, or any related presentation, and SAP SE's or its affiliated companies' strategy and possible future developments, products, and/or platforms, directions, and functionality are all subject to change and may be changed by SAP SE or its affiliated companies at any time for any reason without notice. The information in this document is not a commitment, promise, or legal obligation to deliver any material, code, or functionality. All forward-looking statements are subject to various risks and uncertainties that could cause actual results to differ materially from expectations. Readers are cautioned not to place undue reliance on these forward-looking statements, and they should not be relied upon in making purchasing decisions.

SAP and other SAP products and services mentioned herein as well as their respective logos are trademarks or registered trademarks of SAP SE (or an SAP affiliate company) in Germany and other countries. All other product and service names mentioned are the trademarks of their respective companies.

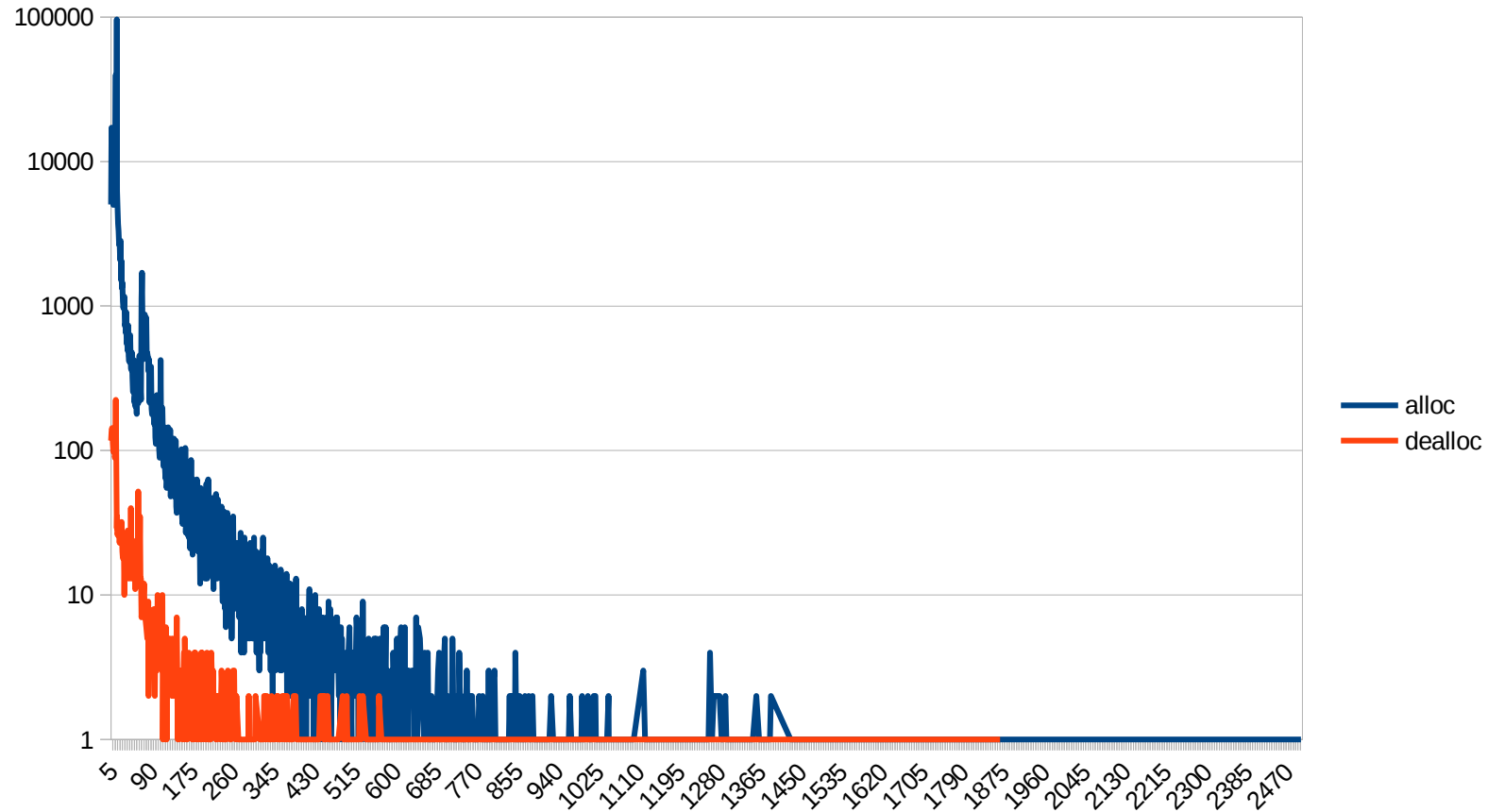
See www.sap.com/copyright for additional trademark information and notices.

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)



- from a wildfly startup

- allocation: only about 1% above 40 words

- deallocation: similar, but higher number of larger blocks

Why bother?

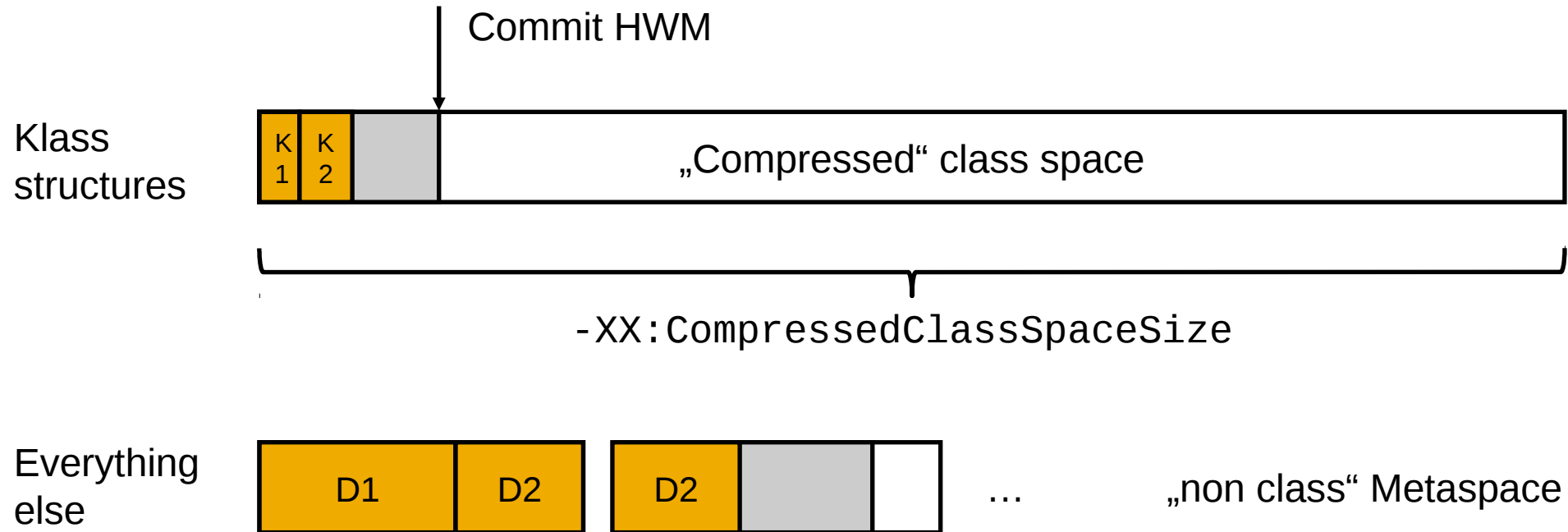
Why not a general purpose allocator?

We think we can do better:

- Arena style allocation is fast and memory efficient.
- We know the size distribution of typical allocations
- Case against malloc in particular:
 - CompressedClassSpace
 - Platform specific limitations (e.g. sbrk hits java heap)

Current implementation

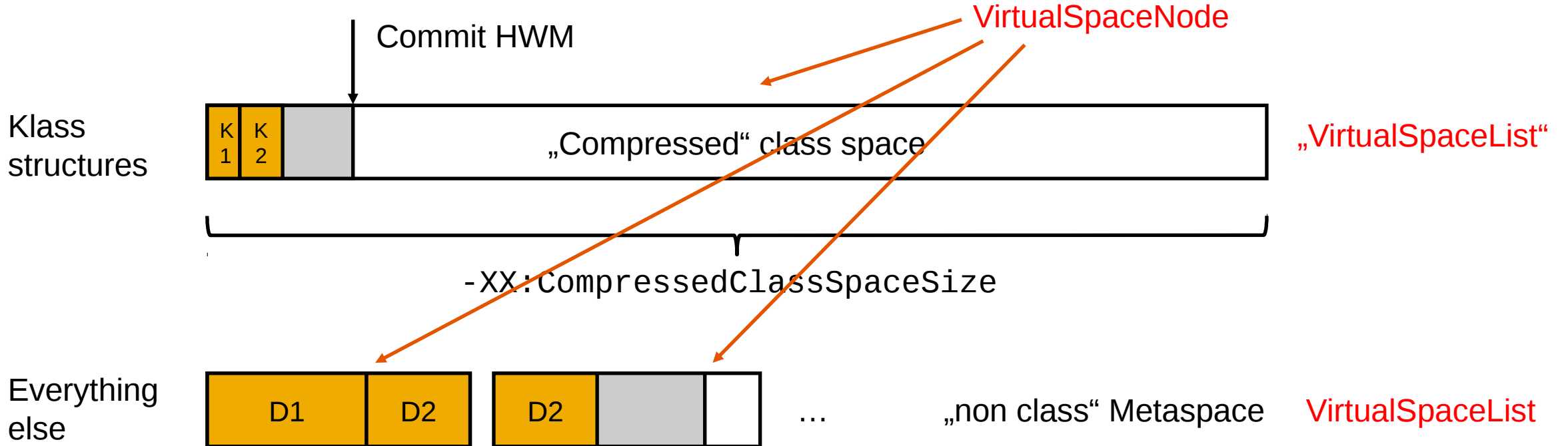
Metaspace has two parts



Sizes per class:

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

Metaspace has two parts

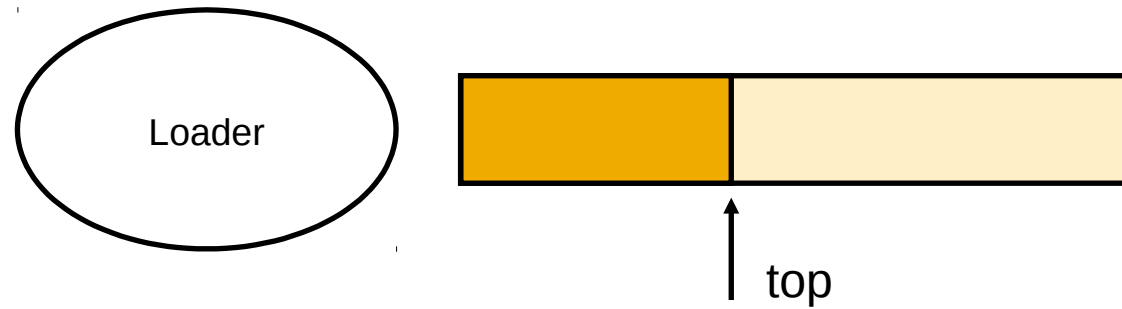


Sizes per class:

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

Current implementation (1)

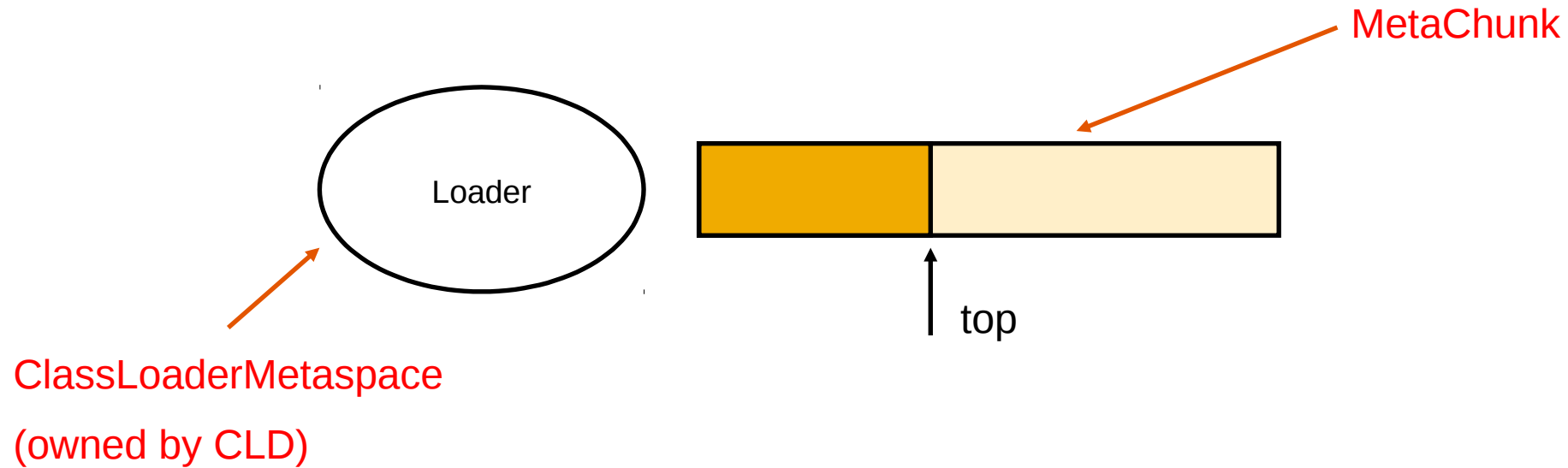
(much simplified)



- Loader owns a chunk of memory.
- Allocates from it via pointer bump.

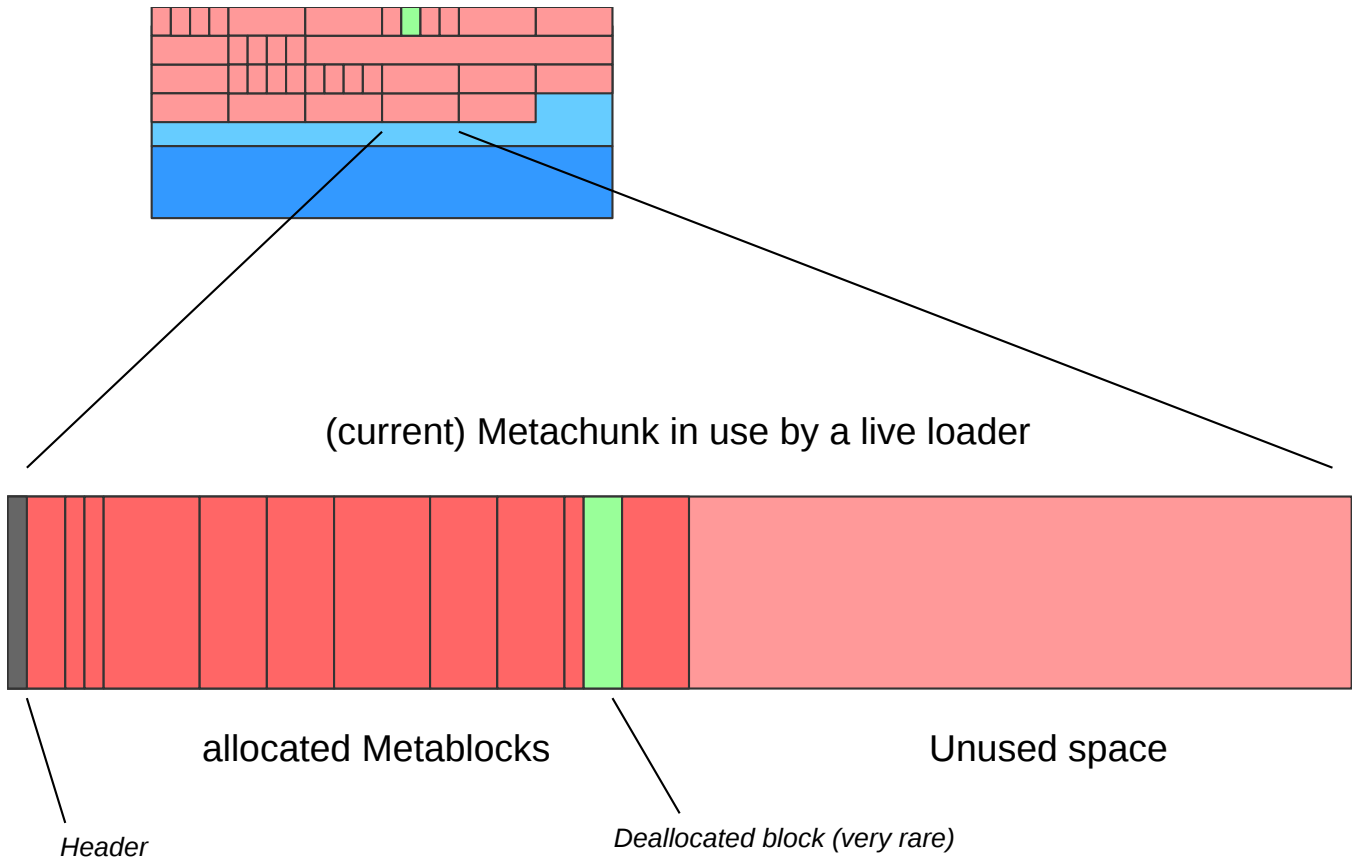
Current implementation (1)

(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.

MetaChunk



Current implementation (2)

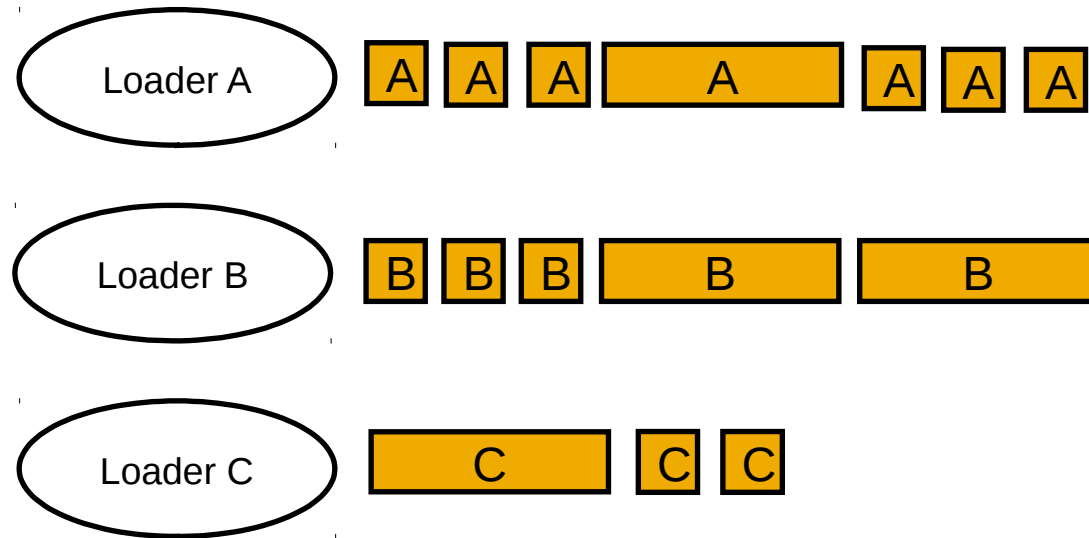
(much simplified)



- 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

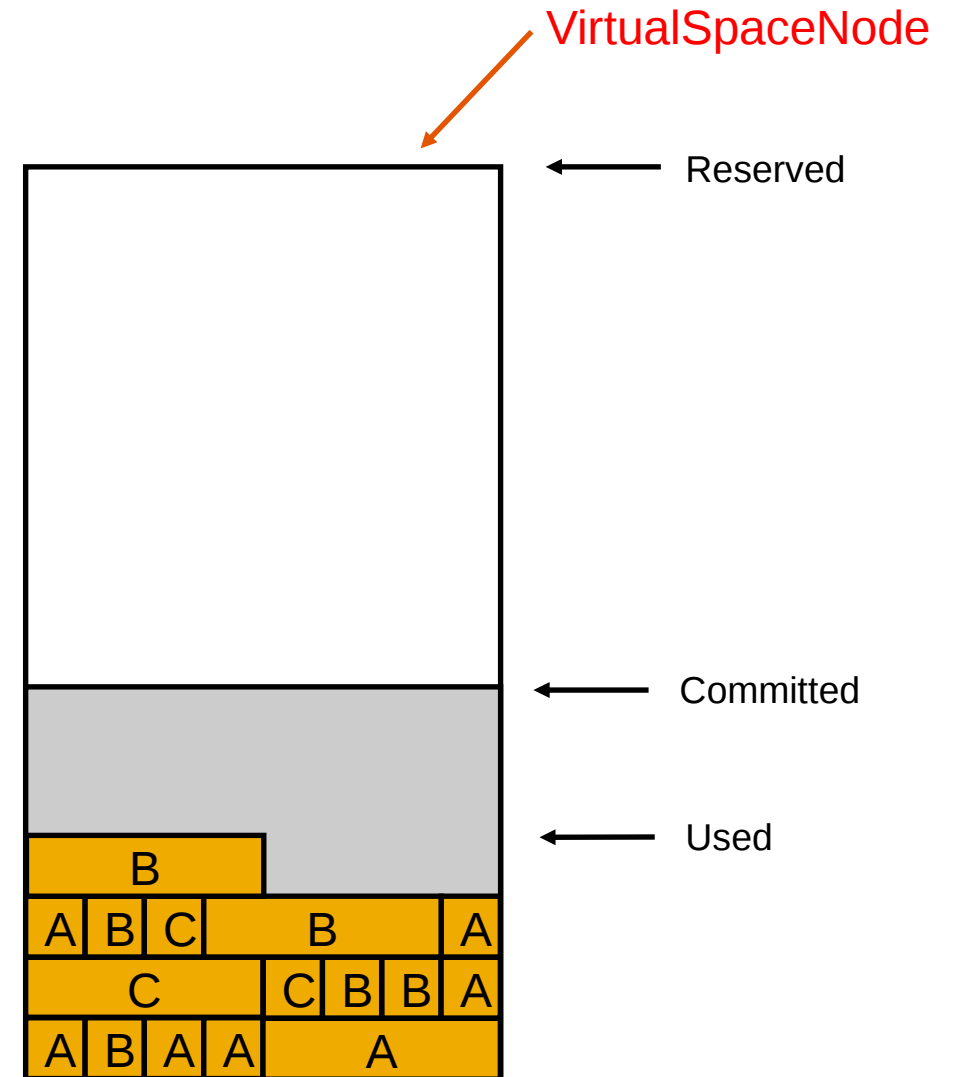
Current implementation (3)

(much simplified)



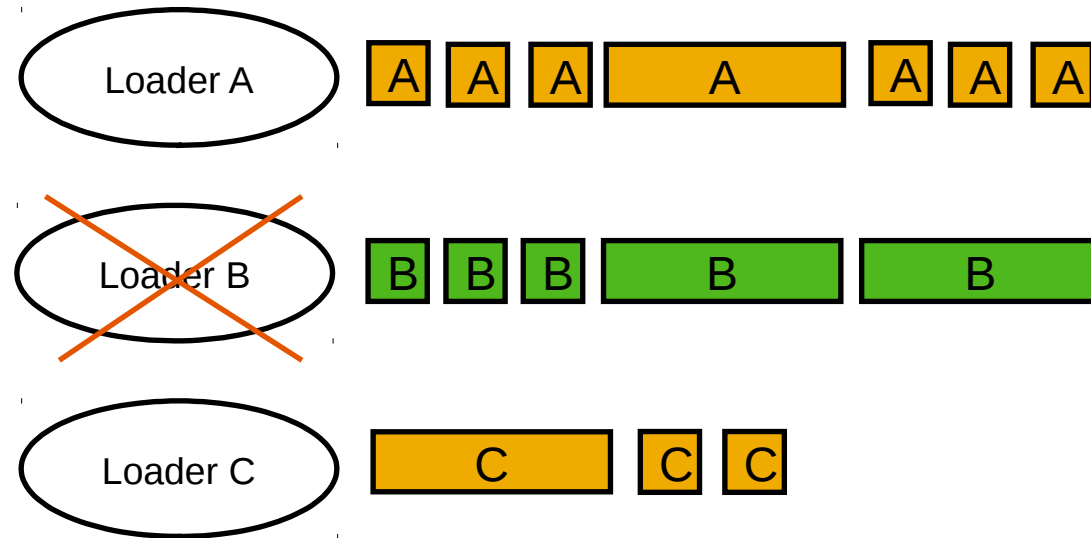
Chunks are carved from **VirtualSpaceNode** as they are allocated.

VirtualSpaceNode is committed on demand, never gets uncommitted.

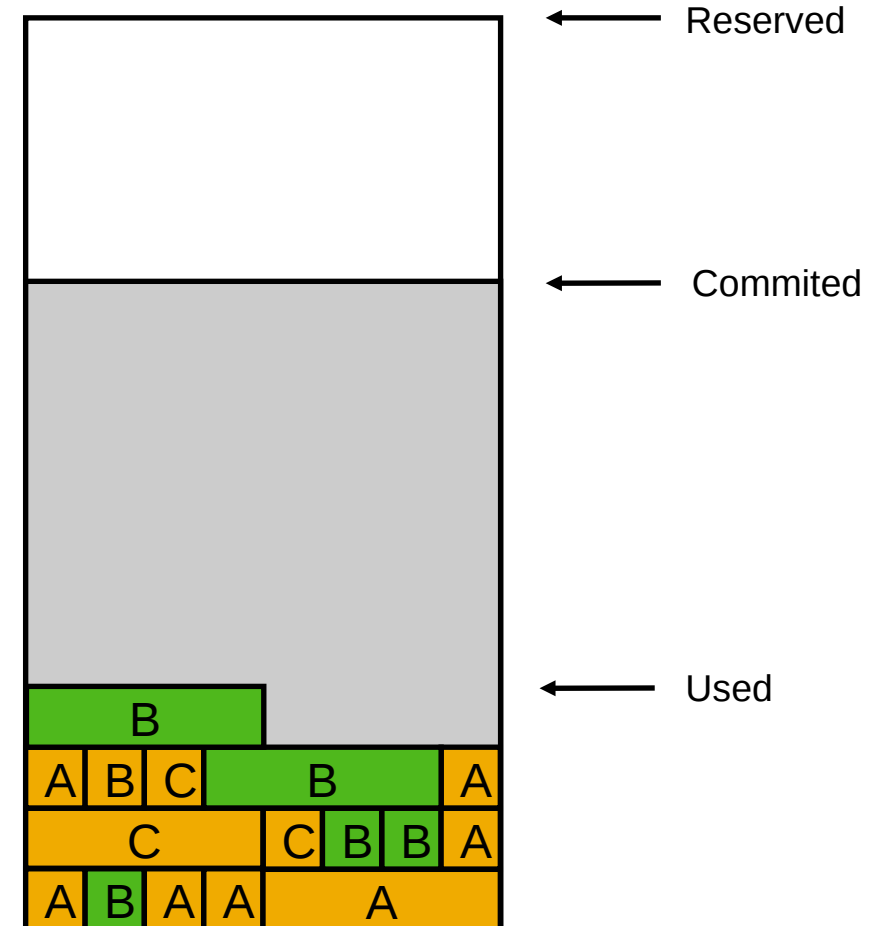


Current implementation (4)

(much simplified)

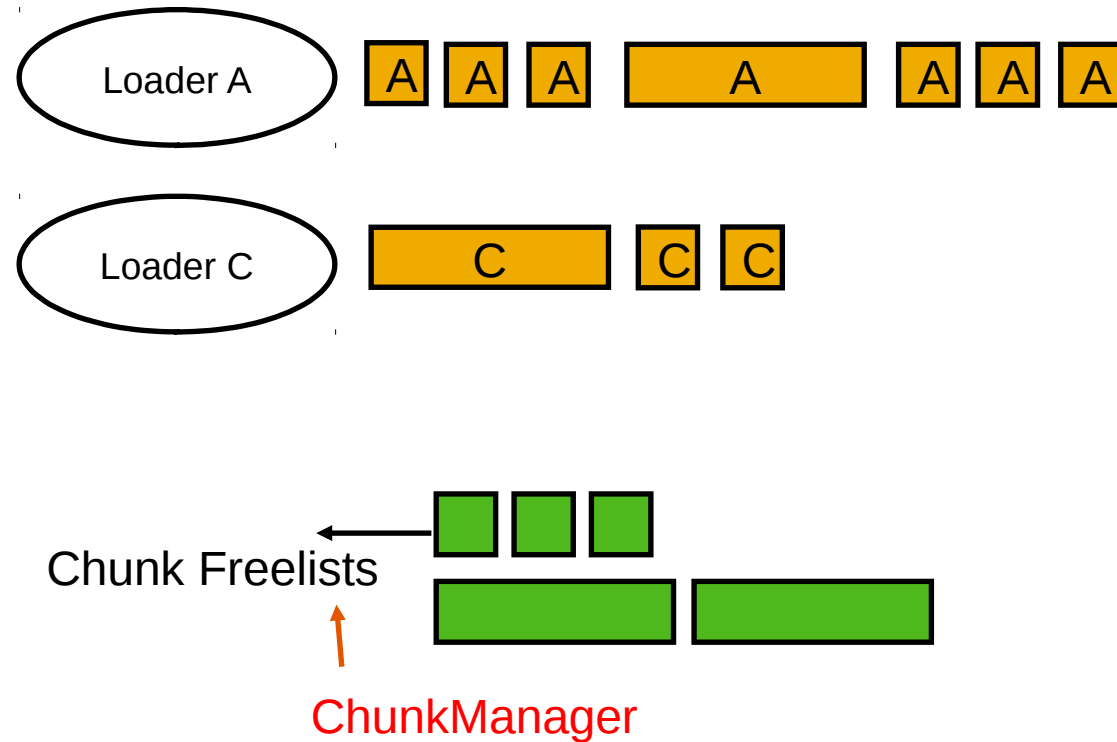


When a loader dies, its chunks are marked as free...

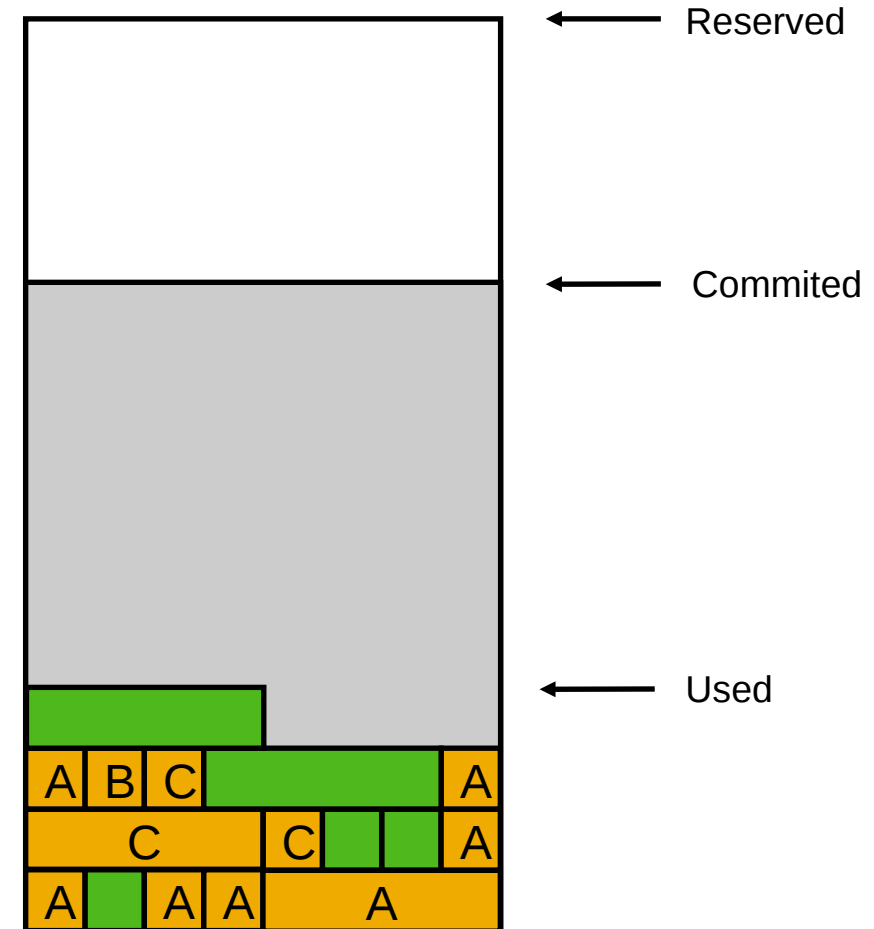


Current implementation (5)

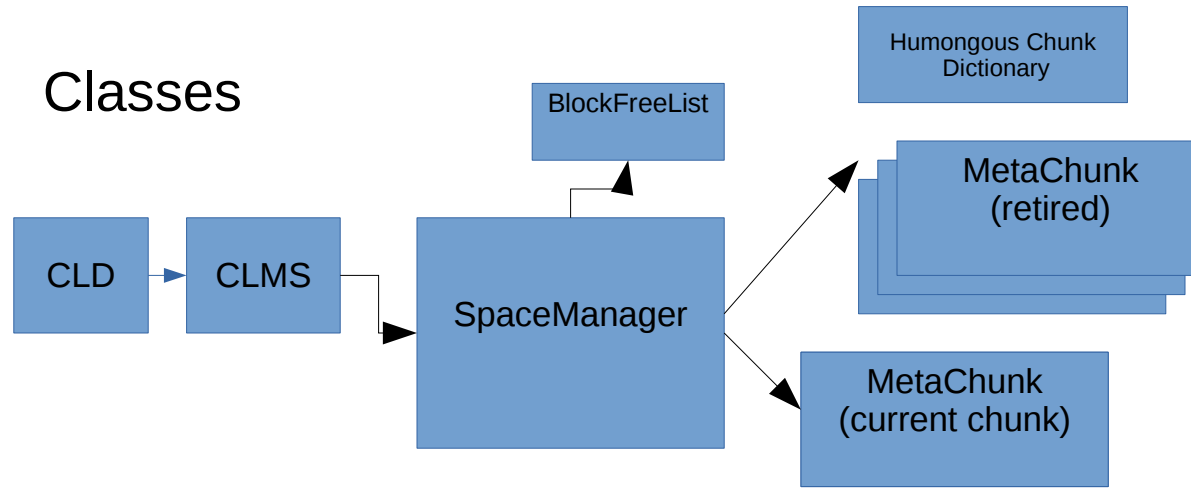
(very much simplified)



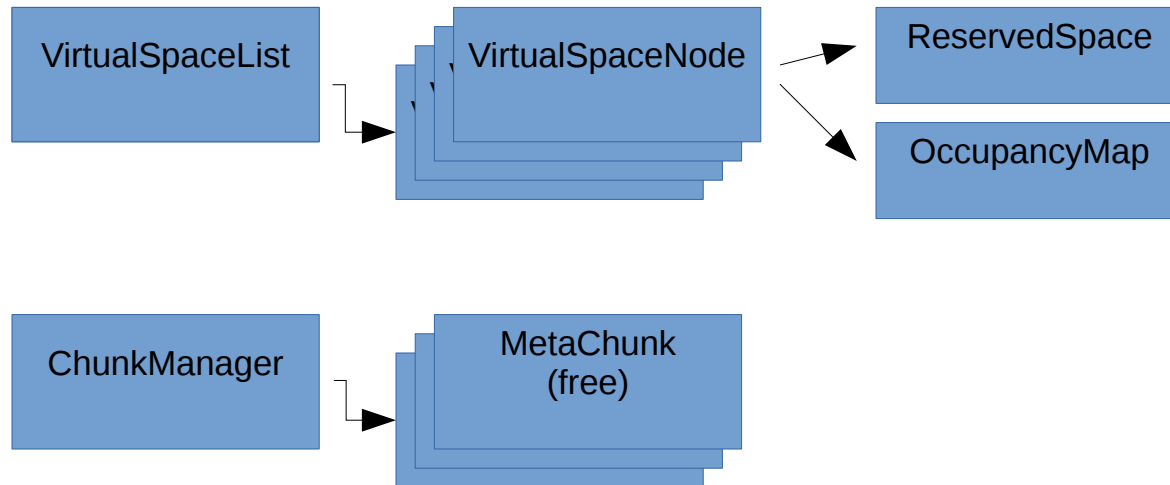
...and added to global freelists (**ChunkManager**), sorted by size. **VirtualSpaceNode** is potentially unmapped.



Classes

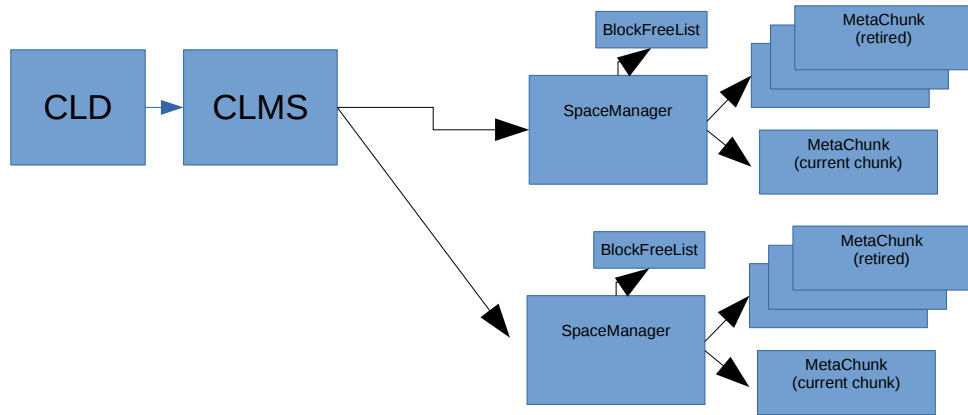


Per class loader



Global

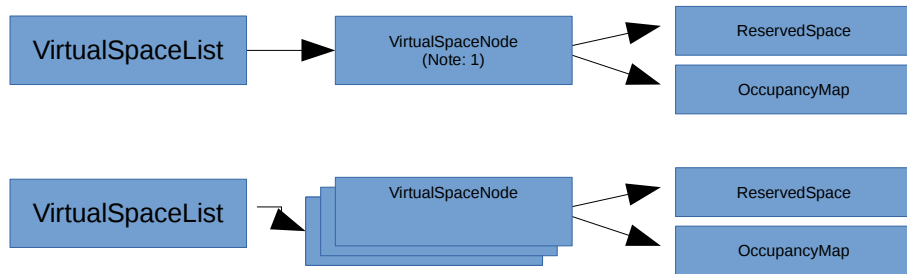
..with CompressedClassPointers



Class metadata

Per class loader

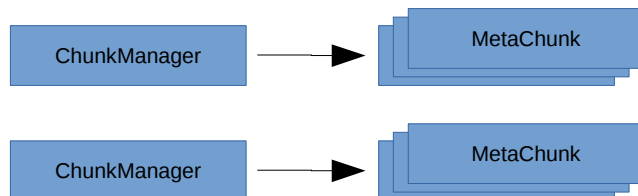
“non-class” metadata



Class VSlist (note: really just one element).

Non-Class VSlist

Global



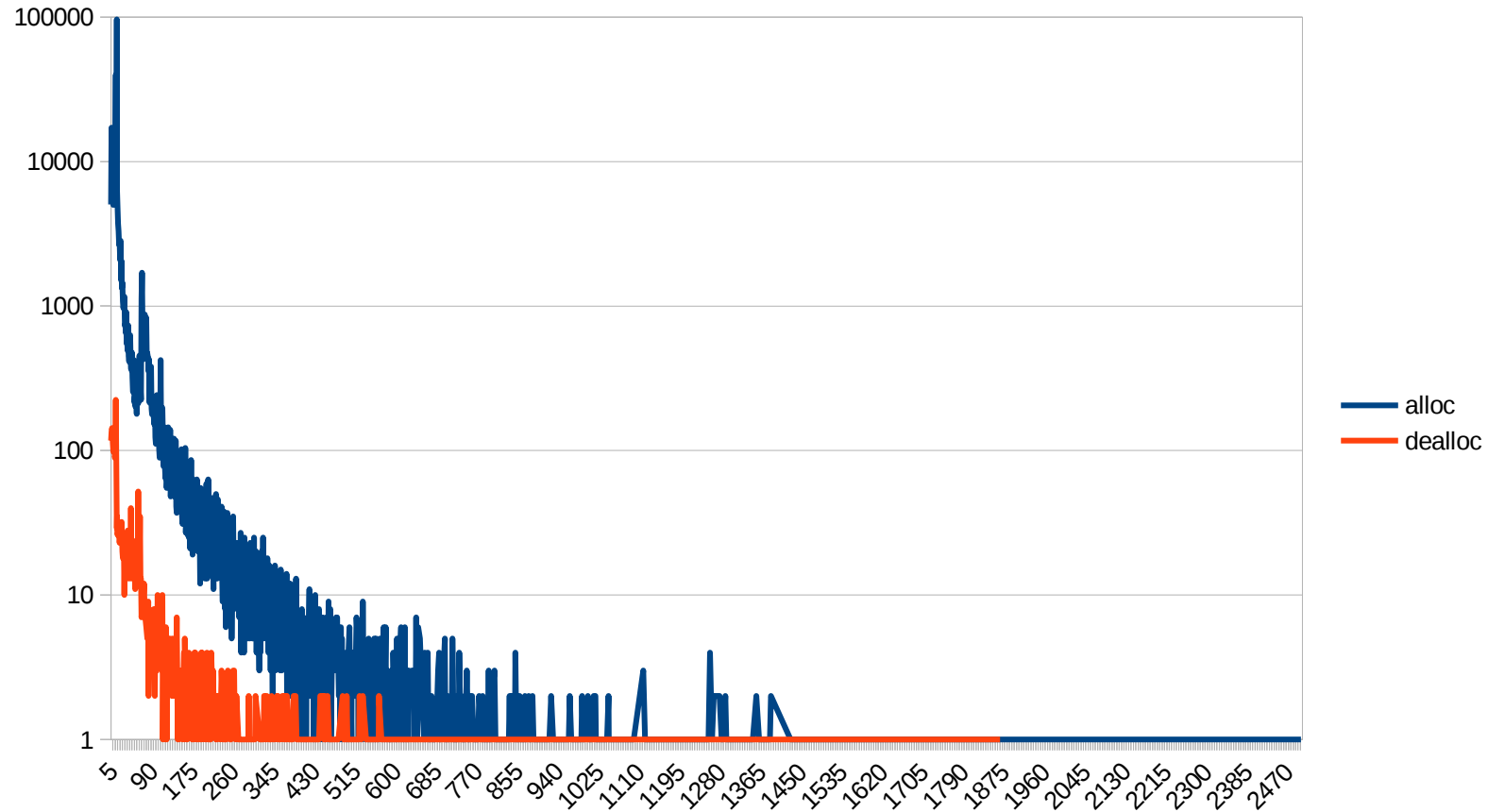
Class ChunkManager

Non-Class ChunkManager

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. But may happen 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)

Metadata allocation / deallocation histogram (blocks/word size)



- from a wildfly startup

- allocation: only about 1% above 40 words

- deallocation: similar, but higher number of larger blocks

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
- Note: blocks/tree nodes in place
- Whats not good:
 - Smallblocks does not search for larger blocks
 - Dictionary: too large blocks are retrieved and inserted back unnecessarily
 - We are afraid and don't use it
 - Complicated code

„Micro Loaders“

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

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

Eat your plate up!

- The “eat your plate up” problem: how large a chunk do we give to the class loader?
 - Too small: high fragmentation and contention on central parts
 - Too large: wasted space
 - Often guesswork and can be wrong
- A function of chunk allocation history
- Micro loaders get small chunks

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

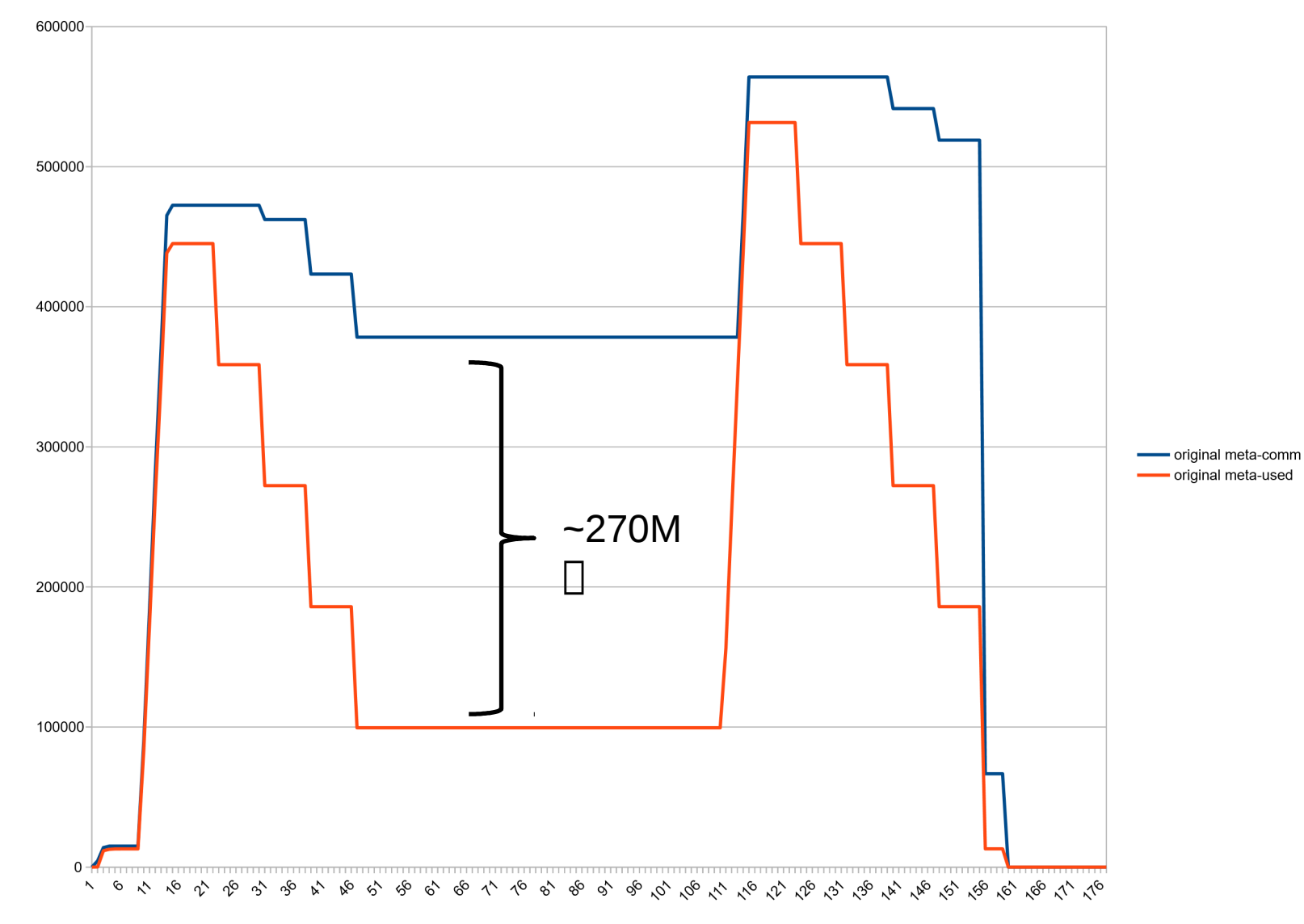
11 – metachunk coalescation

- The „chunk size choking problem“
- JDK-8198423
- Chunks can be merged and split now
 - 4x1K chunks -> 4K
 - 16x4K chunks -> 64K
- Basically the whole thing is now like a crooked buddy allocator
 - Also kind of inefficient
 - 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 :-(

Bottomline: Main problems with 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
- Clean up / simplify code

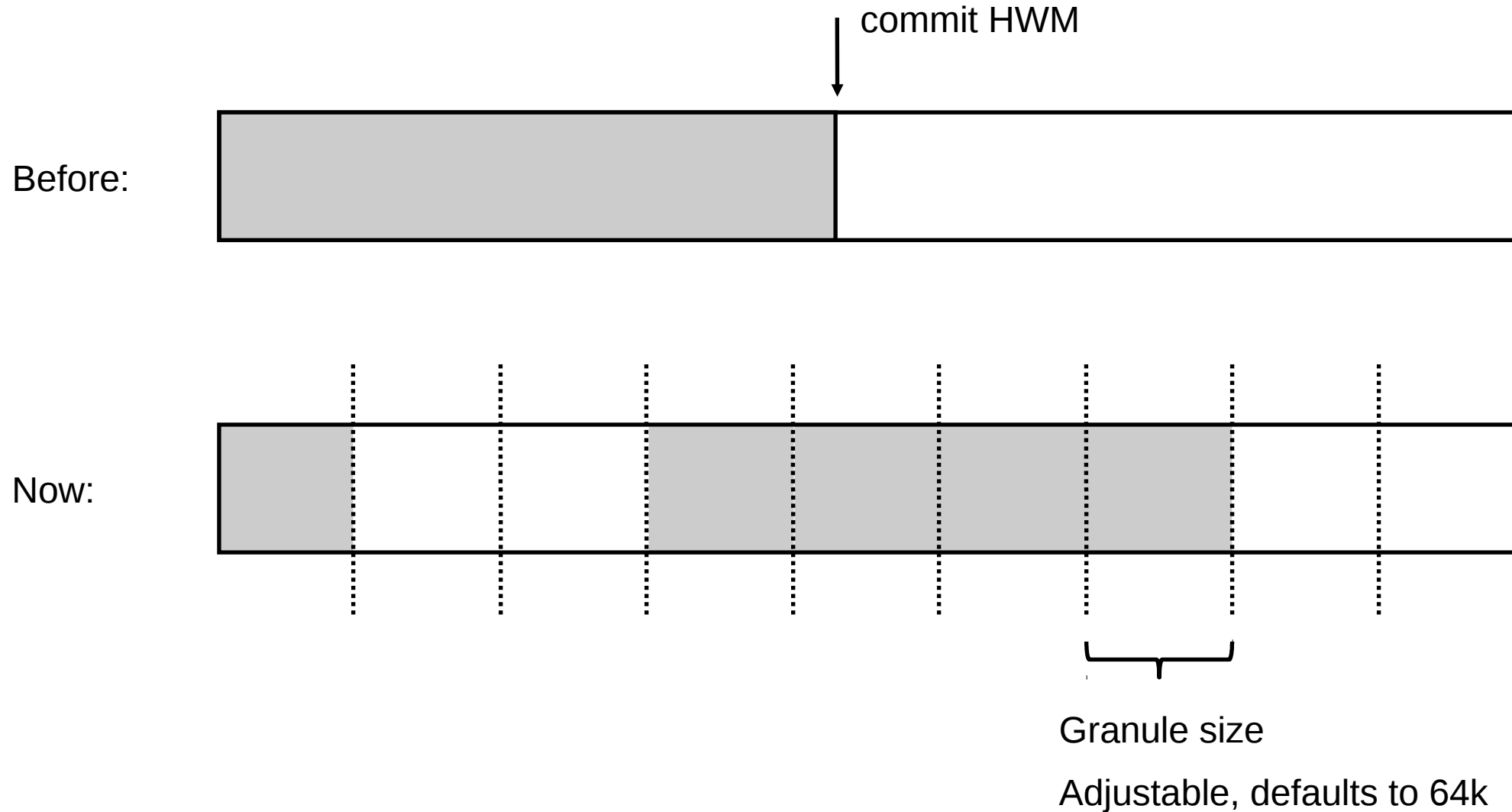
Concern 1: keep number of virtual memory areas low

- (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

Concern 2: uncommit speed

- Matters for Gcs which unload classes in pause
- (Linux):
 - This depends on population of uncommitted area: how many pages in which size had been committed before
 - Hence indirectly on size
 - And also on number of vma s in committed region, although in our case its always 1
- Currently I see no problems in practice, but a fallback plan would be to uncommit concurrently.
 - Not that complicated if we keep locking scheme

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

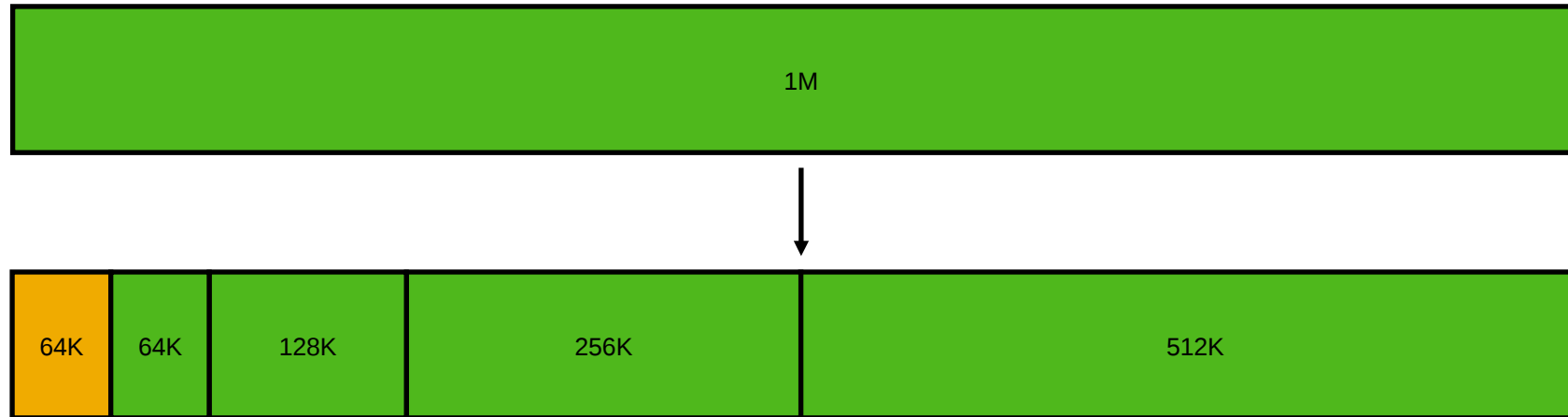
Pow 2 based buddy allocator for chunks

```
// 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
```

Pow 2 based buddy allocator for chunks

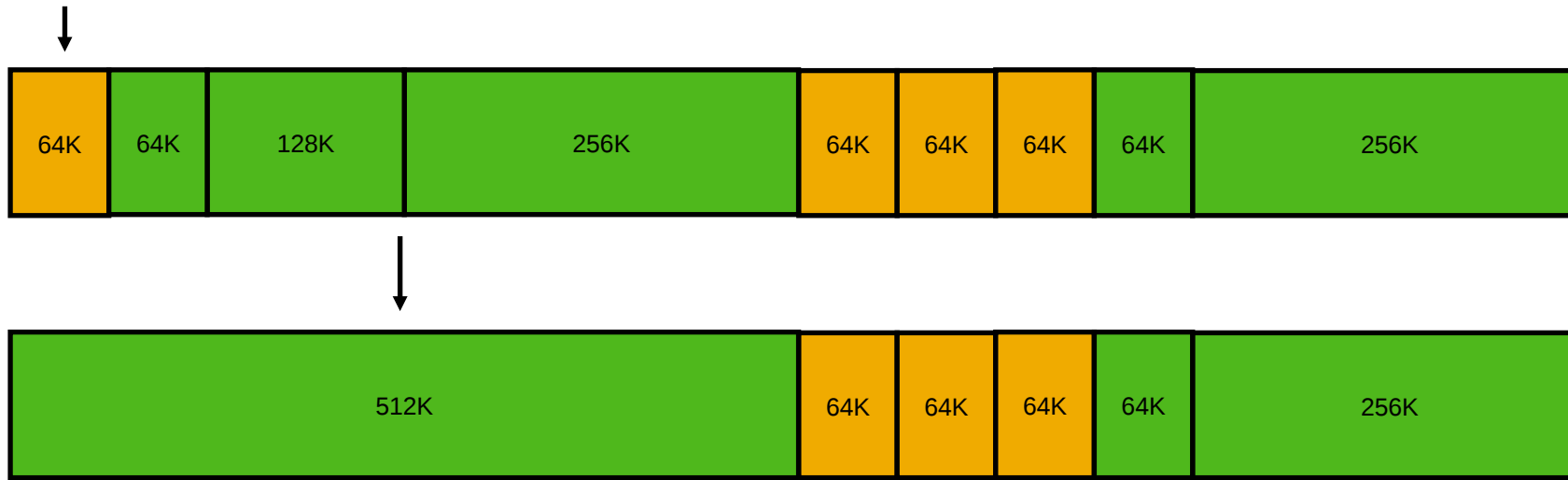
- The whole memory range is now segmented into “root chunk areas” of 4MB size, is aligned at 4MB and gets committed with 4MB grain
- VirtualSpaceNode now only allocates root chunks → much simplified code
- ChunkManager now has 13 free lists, one for each chunk level
- Allocation:
 - To ChunkManager: do we have a chunk of level x?
 - ChunkManager: looks in freelist(x) – yes → return
 - No → look for larger chunk in freelists(x++) and, if found, split chunk. Add splinter chunks back to freelist.
 - If not found: ask VirtualSpaceNode for a new root chunk, and proceed as above

Buddy allocator: Allocation



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

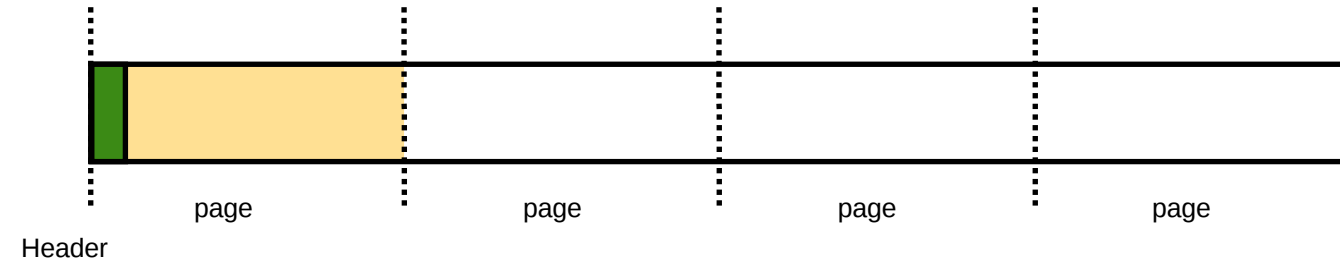
Buddy allocator: Deallocation



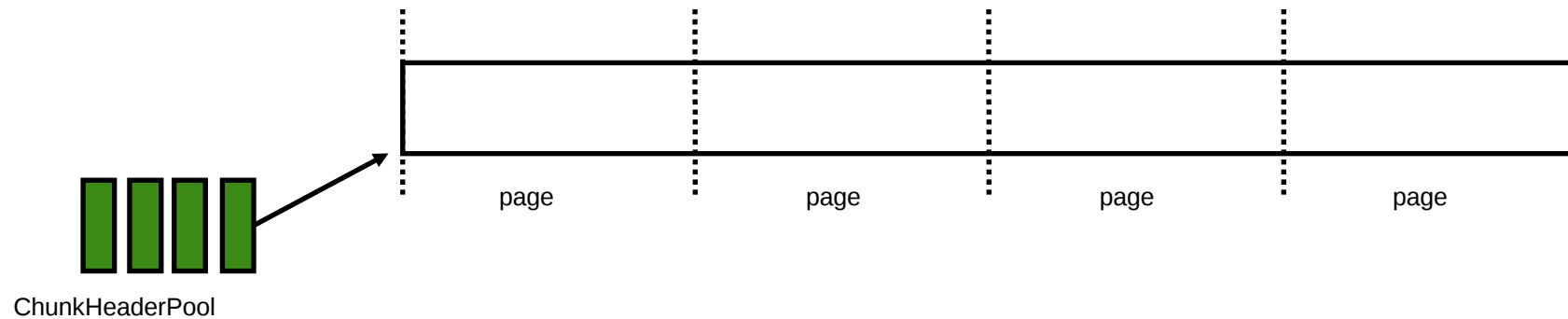
- Mark chunk as free
- If buddy chunk is free and unsplit: remove from freelist and merge with chunk
 - Repeat until root chunk sized reached or until buddy is not free
- Return result chunk to free list

Chunk headers needed to go

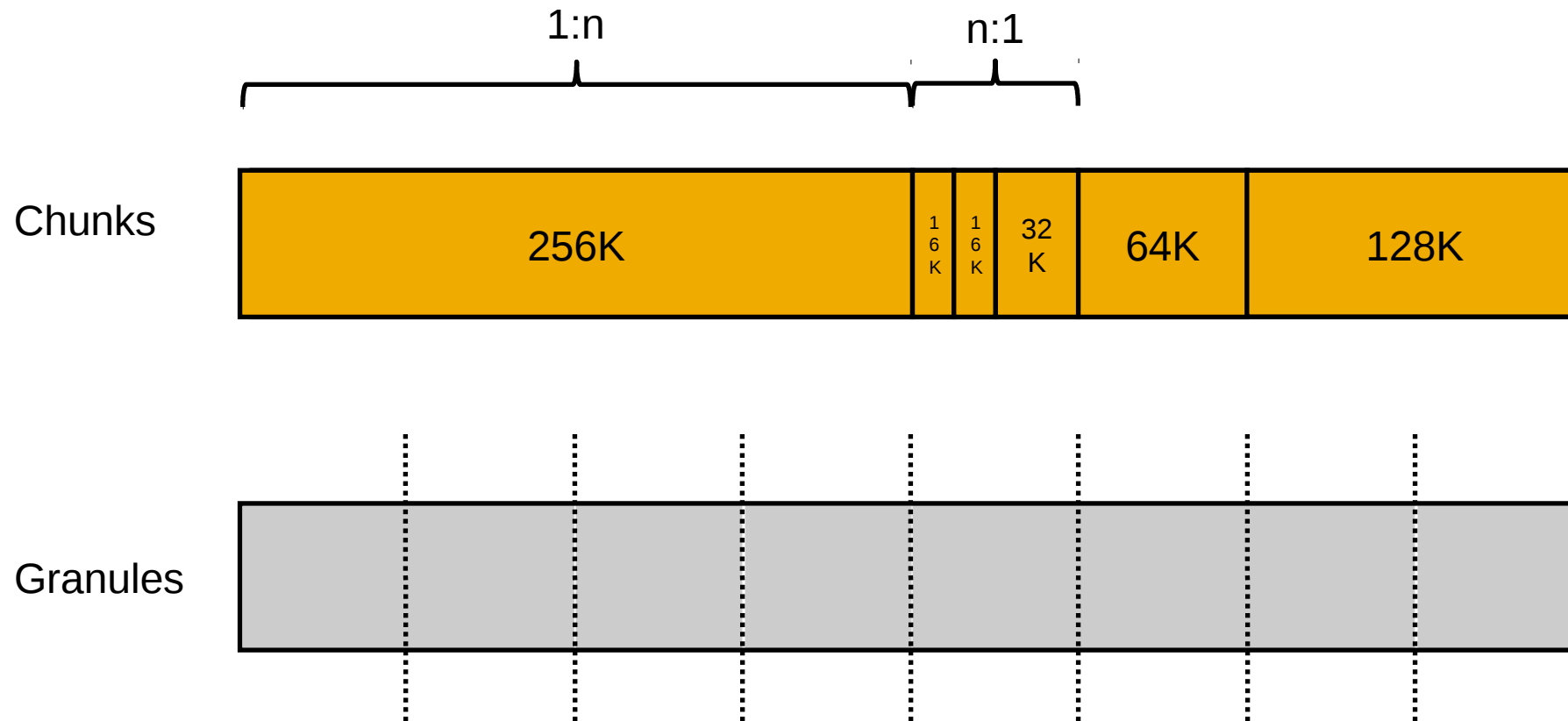
Before



Now

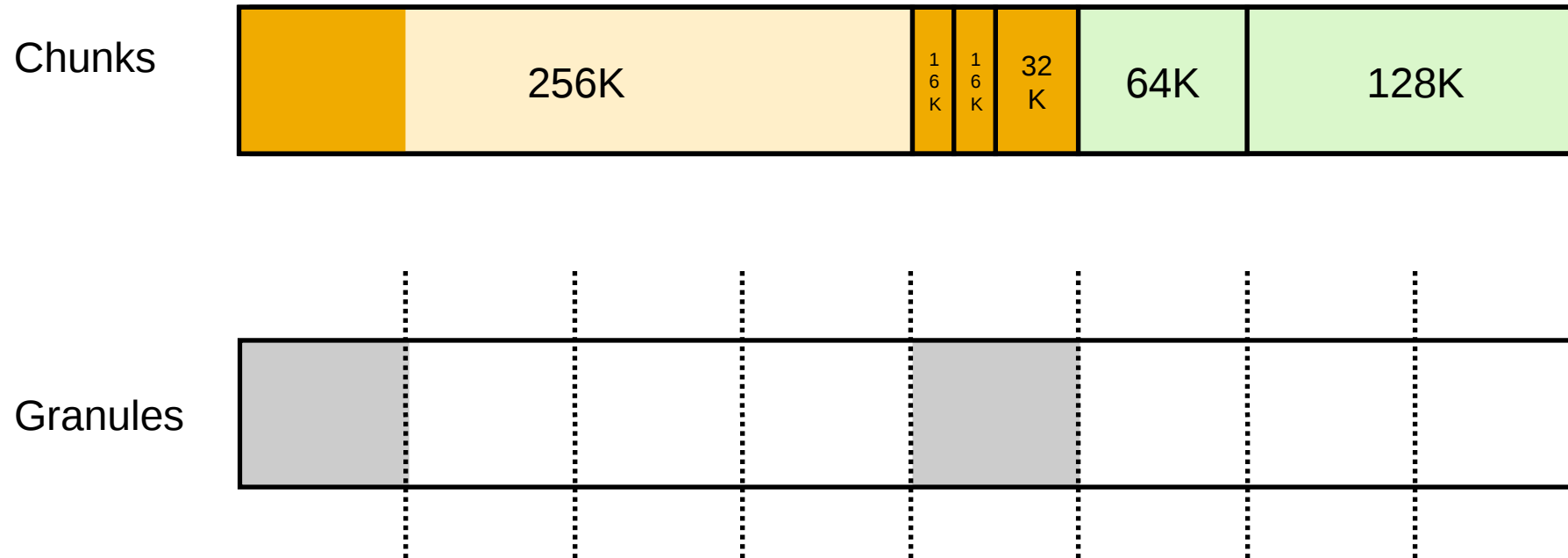


Granules and chunks



- ▮ 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

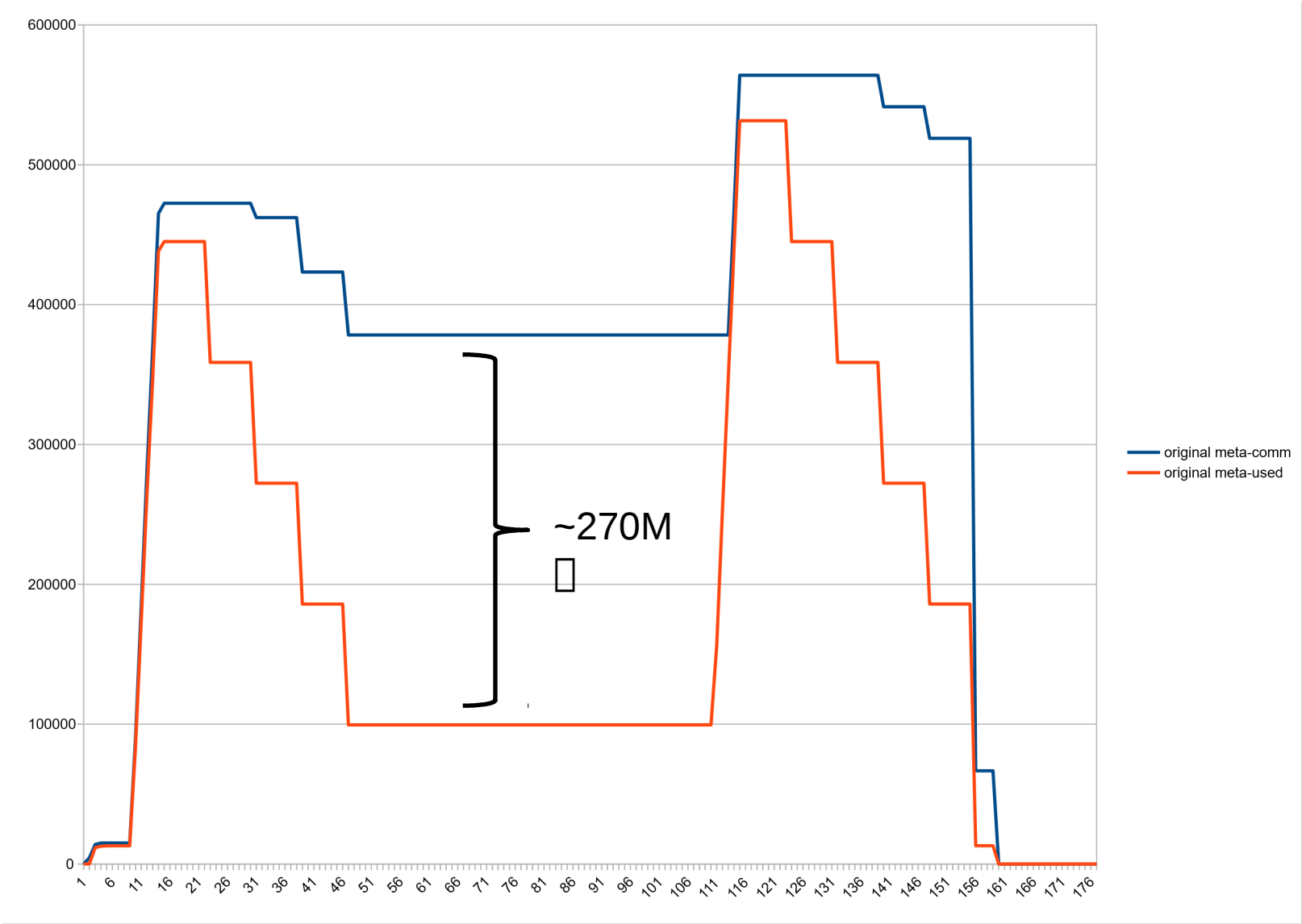
Deallocation handling („LeftOverManager“)

- Combination of bin list (similar to SmallBlocks) and a newly written BST
 - We can get rid of binaryTreeDictionary since now all users are gone
 - The new binlist covers more sizes (atm 32) and searches upward too
 - The new BST is similar to the old one but much reduced in code size and very much simpler.
 - Still unbalanced, making this an RB tree would be nice
 - New BST knows its largest node size.
- We now split blocks where it makes sense.
- Other changes:
 - We call into LeftOverManager for every allocation. This prevents pop buildup.
 - Recent deallocations are just rolled back in the chunk

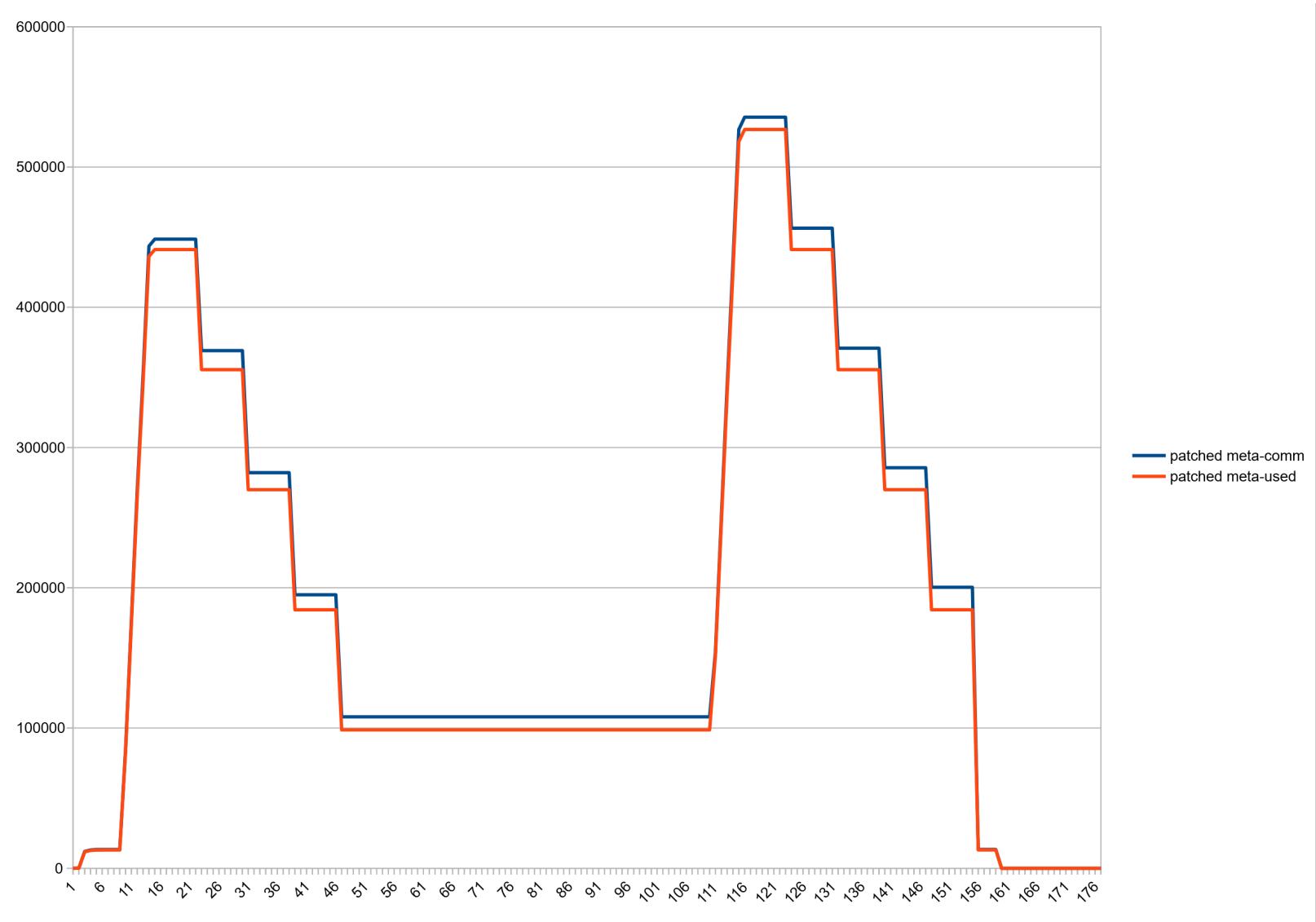
What else changed

- 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.

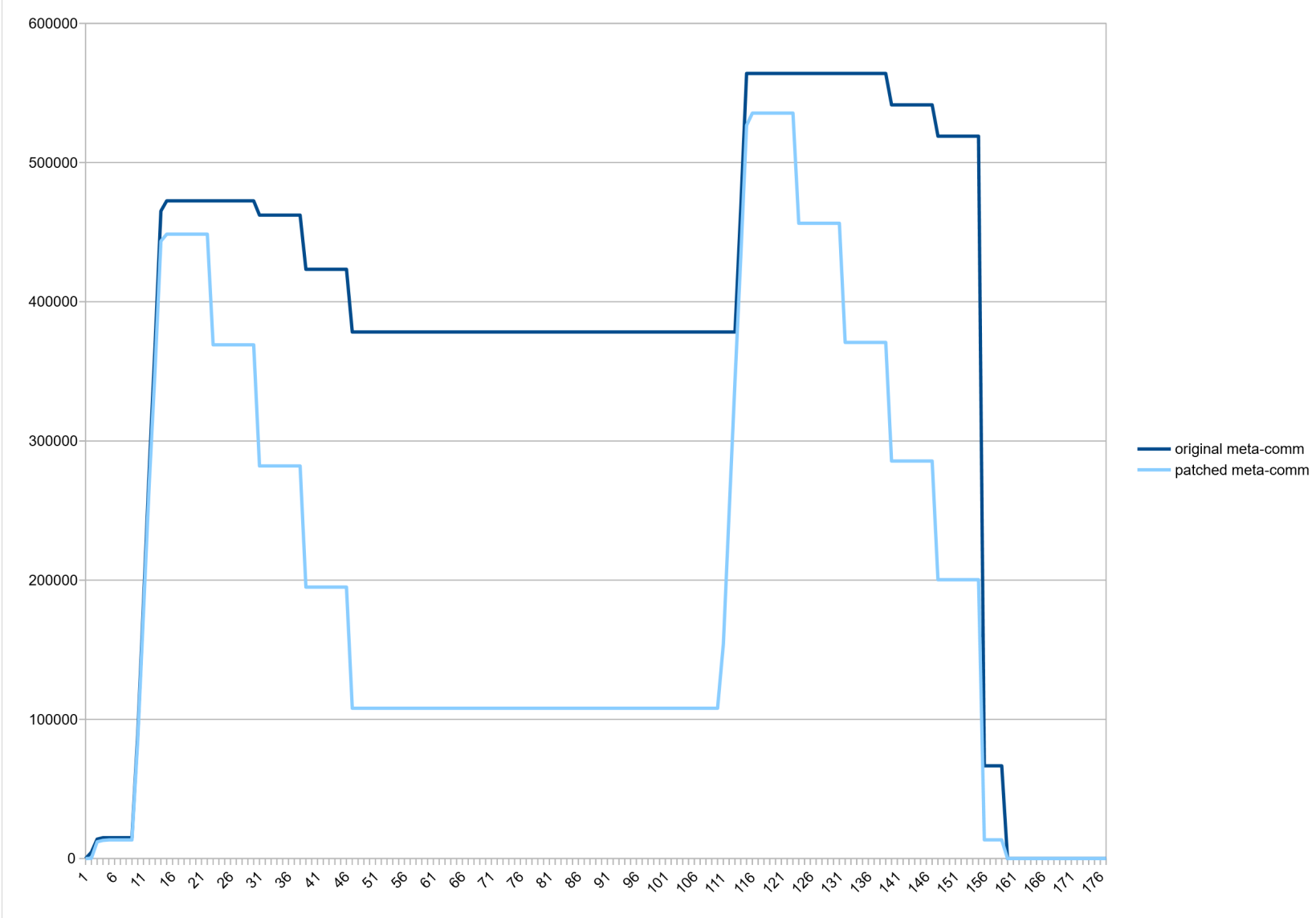
Result: Committed vs used, Stock JDK14



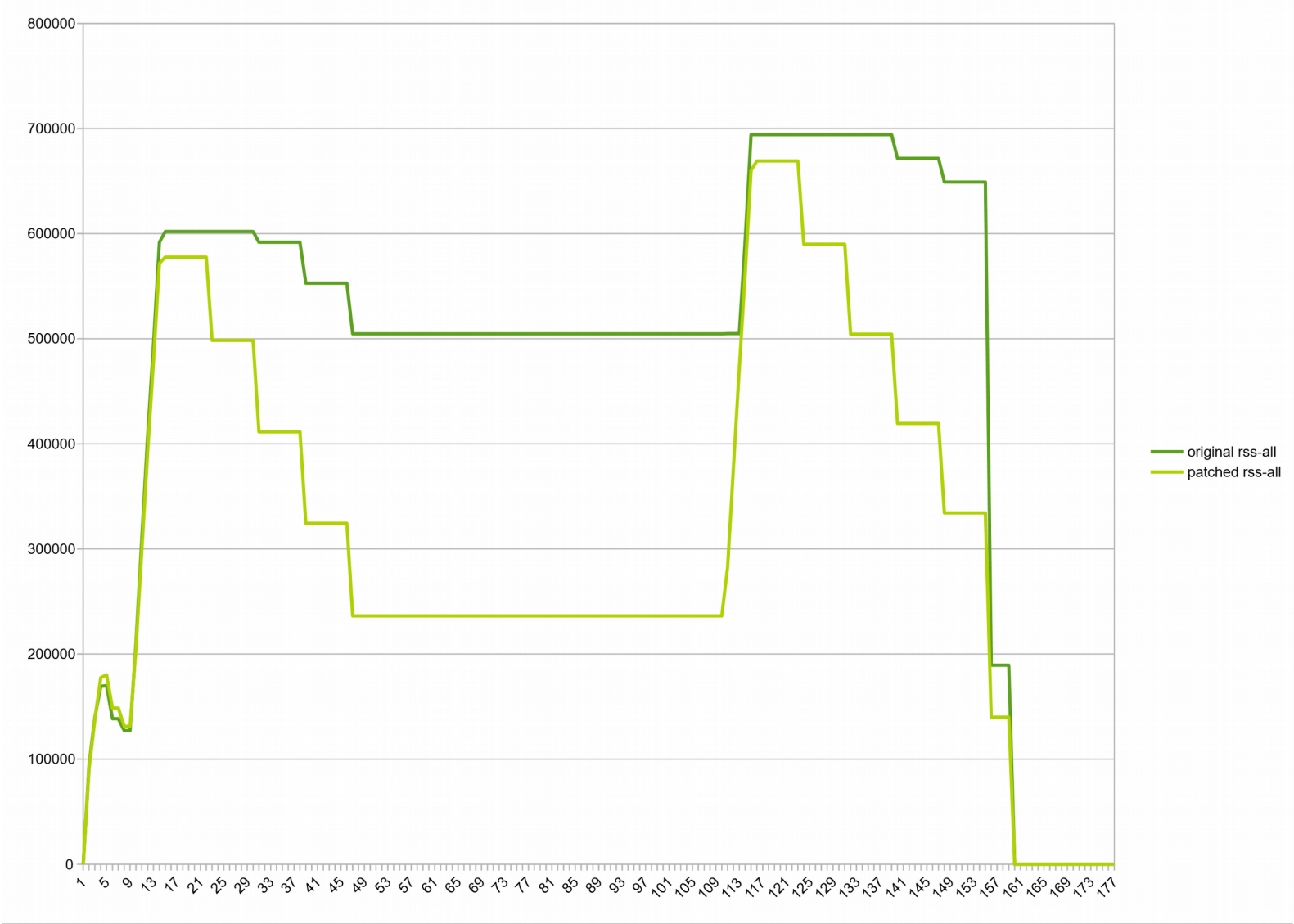
Result: Committed vs used, Patched JDK14



Result: committed Metaspace, Stock vs Patched VM



Result: RSS, Stock vs Patched VM



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%)

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](https://www.instagram.com/tstuefe)

thomas.stuefe@sap.com

[stuefe.de](https://www.stuefe.de)