A new and better Metaspace
for the OpenJDK

Thomas Stüfe, SAP
09, 2019

PUBLIC

Why?

History

“Java uses too much memory” ☹

- Volker’s Memory initiative
- Metaspace may consume a large part of the non-java-heap memory
- I kind of “inherited” the coding -> no point in waiting for others
- Encouragement from Oracle folks at FOSDEM 19
  – And also from jruby folks (Charles Nutter).
- Lets do a clean room reimplementation!
  – 11 Metaspace coalescation patch okay but far from perfect

Metadata Life Cycle Basics

- Metadata get (usually) allocated when a class is loaded
- Its lifetime is bound to that of the loading class loader
  – That loader is only collected when all its classes have been collected
  – -> metadata storage lives as long as there is still one “sibling” class alive
- Exception: premature deallocation (uncommon)

Why a home-grown allocator?

why not just malloc?

- The obvious reason: Compressed Class Space.

- We know things malloc don’t:
  - arena like allocation behavior
    - can throw away all allocations when CL dies.
    - no need to track individual allocations
    - per-CL pointer bump allocation
  - We know the typical allocation sizes.
  - Allocations owned by class loader. Typically no global lock needed.

- malloc has limits (depends on OS)

Metaspace – how it works

- Global list of mmaped regions (VirtualSpaceNode)
- Region is split up in chunks which are handed to class loaders
  – 4 kind of chunks: 1K, 4K, 64K
- Loader pointer-bump-allocates from chunk until full, then gets next chunk from VirtualSpaceNode
  – Keeps list of chunks.
- When Loader dies, all its chunks are returned to central freelist.
- Premature deallocation is possible but uncommon
- Chunks may be split/merged since 11

11 – metachunk coalescation

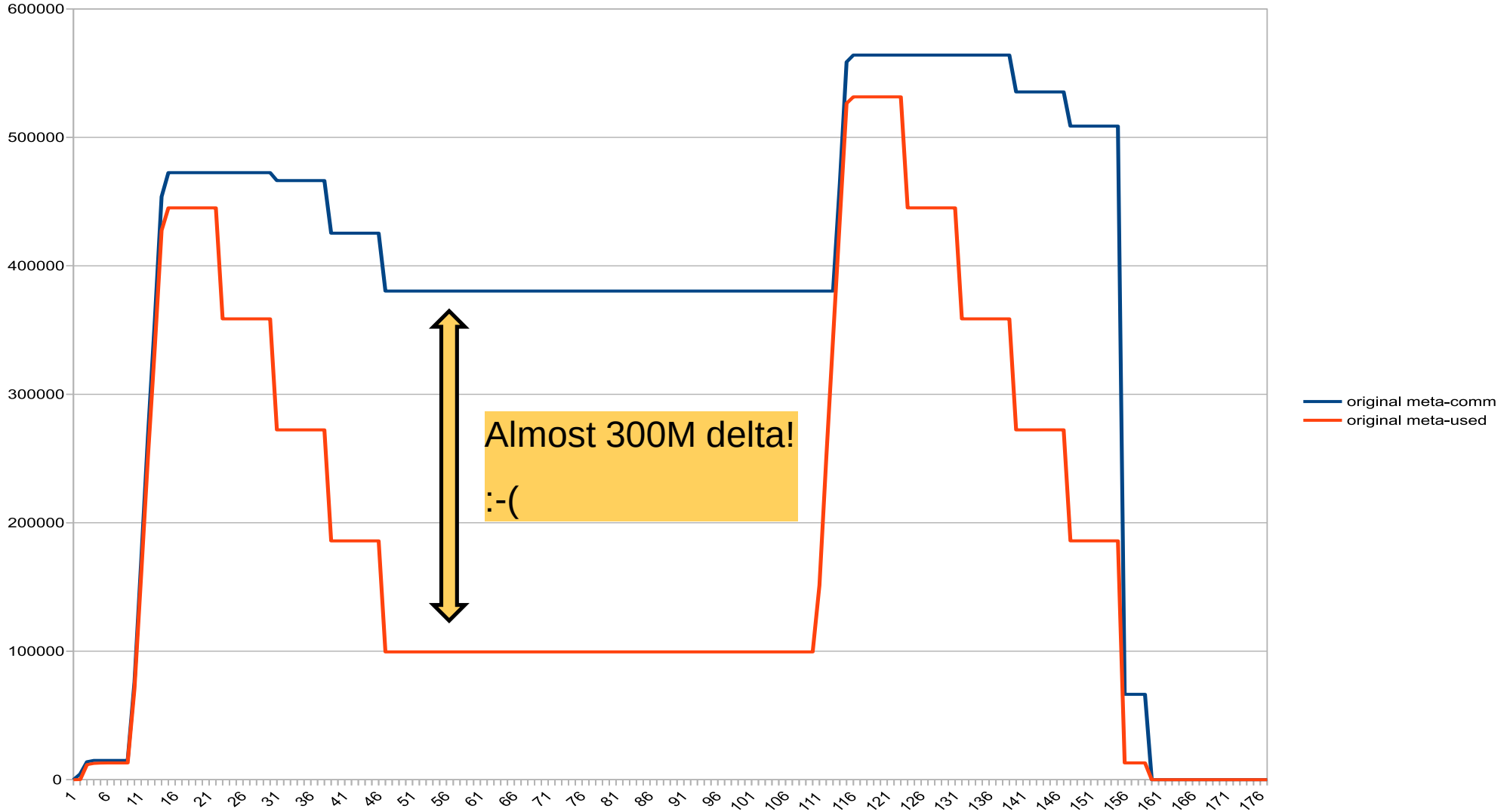
- The „chunk size choking problem“
- JDK-8198423
- Chunks can be merged and split now
  – 4x1K chunks -> 4K
  – 16x4K chunks -> 64K
- Basically the whole thing is now like a crooked buddy allocator
  – Also kind of inefficient
  – But it solved the problem
  – But I was afraid to touch too much, so the whole patch is one gigantic band aid
  – Ugly and difficult to maintain :-(

Problem: bad reclaim rate

- Reclaiming mechanism is insufficient. Easily defeated by fragmentation.
- Result: Freelists can get HUGE! (as in, up to 70% of metaspace)
- -> Metaspace is not really elastic at all.

Bad reclaim rate (Example)

metaspace used vs committed



Problem: bad reclaim rate (Example)

```
$ jcmd  de.stuefe.repros.metaspace.InterleavedLoaders
VM.metaspace
6918:

<cut>

Waste (percentages refer to total committed size 404,82 MB):
              Committed unused:    116,00 KB ( <1%)
        Waste in chunks in use:      2,95 KB ( <1%)
         Free in chunks in use:      6,41 MB (  2%)
     Overhead in chunks in use:    219,69 KB ( <1%)
                 In free chunks:    275,21 MB ( 68%)
Deallocated from chunks in use:      1,29 MB ( <1%) (2227
blocks)
                        -total-:    283,24 MB ( 70%)
```

Problem: Intrachunk waste

Leftover space in chunks, unused, but owned by Class Loader

- after a chunk gets retired
- in the current chunk after CL stops loading classes
- Problem mostly for for "micro-loaders" - loaders which load only one or a few classes (MH, Lambdas, Reflection, jruby)
- Usually 2-4% but can get annoyingly large (e.g jruby)

Reimplementing Metaspace

What stays the same

We keep the same:

- Global list of mmaped regions; region gets carved into CL-owned chunks; chunk freelists
- We still have two “spaces”, non-class (which is still a space list) and class
- Switches should keep meaning:
  – MaxMetaspaceSize and CompressedClassSpaceSize
  – no limit should trigger where it would not trigger before.

What is different

- **A different chunk geometry:**
  - **Buddy allocation** scheme
  - Way more chunk sizes; 1K up to 4M in pow2 steps.
  - 4M chunks are “root chunks”

- **Chunks get committed and uncommitted on demand**

- Got rid of humongous chunks :-)
- Chunks can grow in-place
- Deallocation can happen in-place
- Chunk headers separated from Metaspace

Buddy allocator

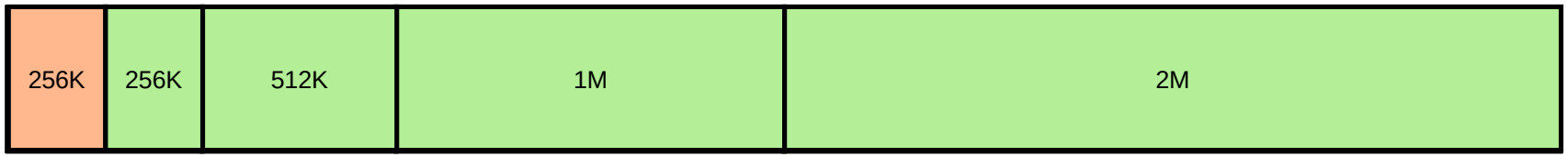
- Chunk sizes from 4MB (“root chunk”) … 1K (smallest) in pow2 steps
  - *Why 1K? -> larger than 99% of InstanceKlass*
  - *Why 4M? -> comfortably larger than the largest possible InstanceKlass*

- 13 sizes (“levels”). Root chunk is level 0.

- 13 freelists.

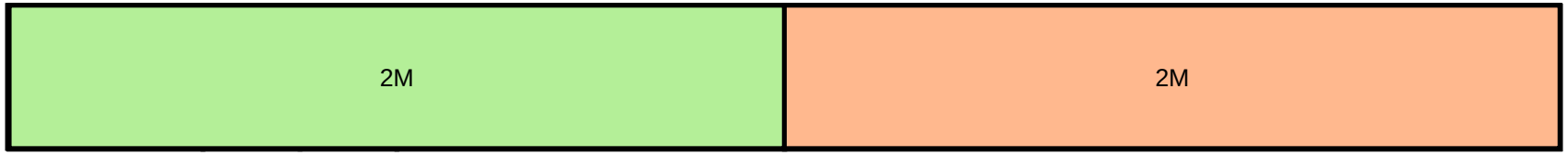
- Implementation via chunk chain.
  - A chunk split is an insert op into a chunk header chain.
  - A merge is a remove op from a chunk header chain.
  - Pretty cheap to implement. Costs 2 pointers per chunk -> 64K for a fully fragmented 4MB range.

Buddy allocator: Allocation



- Remove root chunk from freelist
- Split split split
- Return the result chunk; all other splinters go back to the freelists

Buddy allocator: Deallocation



- Chunk becomes free:
- Merge with buddy if buddy is free
- Rinse repeat

Chunks can now grow in place :-)

If a chunk is too small to satisfy an allocation, instead of allocating a new chunk, we can attempt to grow in place.

- a chunk can grow in place if it is the leader of the buddy pair and the follower chunk is free
- reduces intra-chunk waste. Works best if a single CL loads undisturbed by other loaders.

Buddy allocation: What are the advantages?

- Good defragmentation, also on longer runs.
  - Improved coherence of free areas -> good for uncommitting
- less intrachunk waste because we have more chunk sizes to choose from.
- less intrachunk waste because chunks can grow in place.
- standard algorithm, easy to understand -> lower long term maintenance
- Simple and cheap to implement

New committing/uncommitting scheme

- Chunks can be committed on demand
  – They can start out completely uncommitted and grow as needed
  – Good for large chunks, e.g. BootLoader

- Chunks can be uncommitted when free
  – E.g. Upon return to the freelist
  – or after a metaspace gc purge

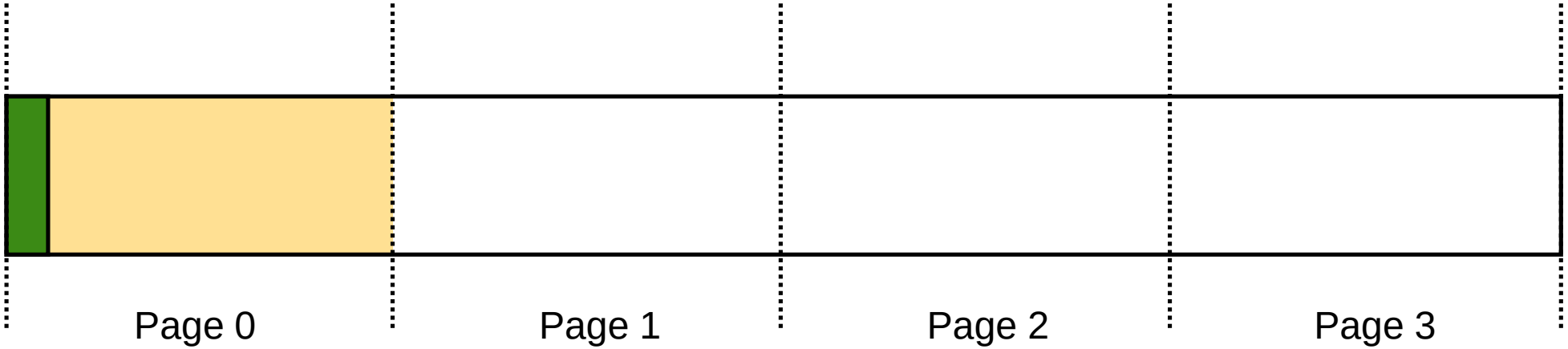
- Hence *Metaspace is not committed bottom-up anymore but may be “checkered”*

Problem: virtual memory fragmentation

Having many small virtual memory areas (eg committed and uncommited ranges) can be counterproductive

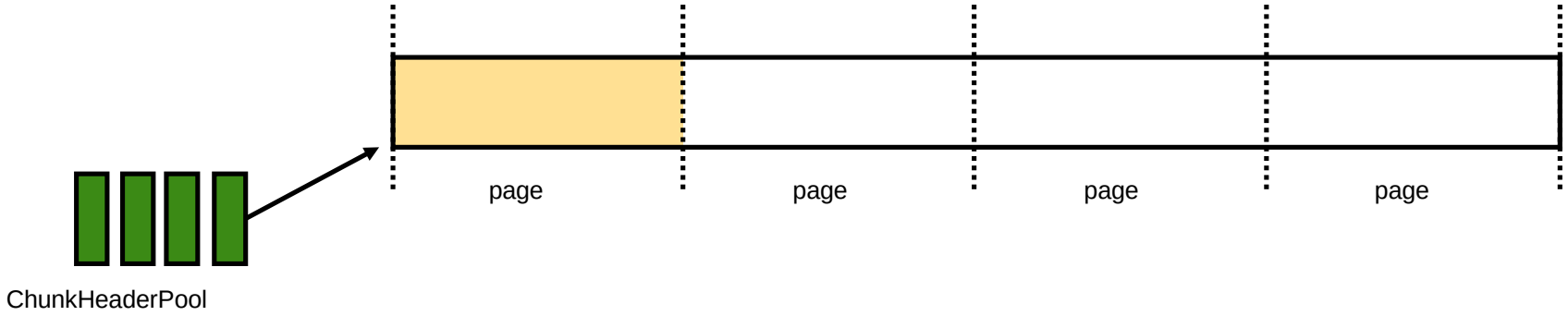
- see e.g. /proc/pid/map
  – E.g. java thread stacks :-/
- Mapping limit! vm.max_map_count
- Note: Linux kernel merges regions with the same attributes together
- So: we must not uncommit in a too fine granular fashion
  – ->uncommit larger areas, not single pages
- So: Chunk headers are a problem!

Chunk headers were a problem.



- Page 0 needs to stay committed, since it contains the chunk header.
- Annoying because
  - Header is way smaller than a page – waste of memory
  - It fragments the virtual memory range, since neighboring uncommited chunks cannot be fused at OS level – there is always one page commited between each chunk

Solution: chunk headers and payload are separated



- Headers now live in an outside pool
  - a simple slab allocator with freelist
  - Also better locality when walking chunk lists
  - One added dereference :/ but I pay for it by removing one somewhere else ☺
- Now we can freely uncommit chunks fully; neighboring free uncommitted chunks are merged by OS vm manager

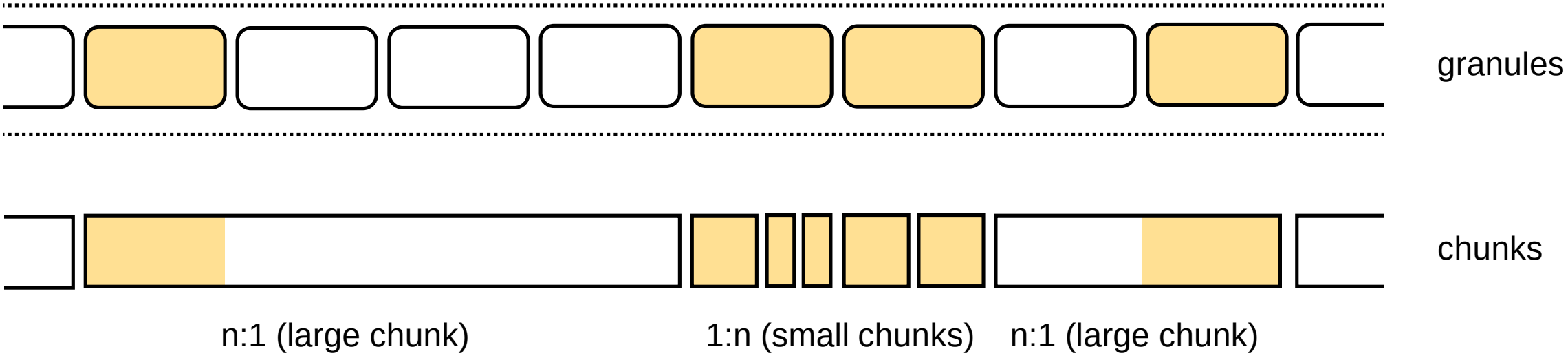
Commit Granules

Don’t commit pages. Commit larger units.

That unit is the “commit granule”.

- Address range is segmented into homogenously sized commit granules.
- Commit granules are committed / uncommitted as a whole only
- Bit mask keeps track of which granules are committed
- Default size 64K

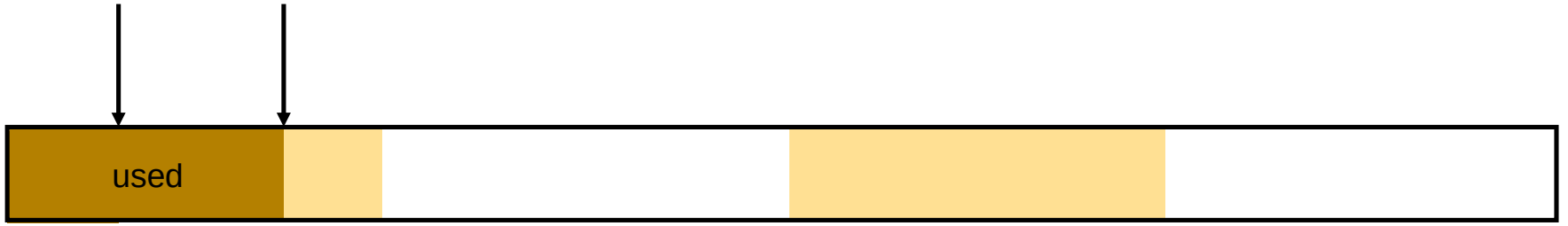
Commit granules



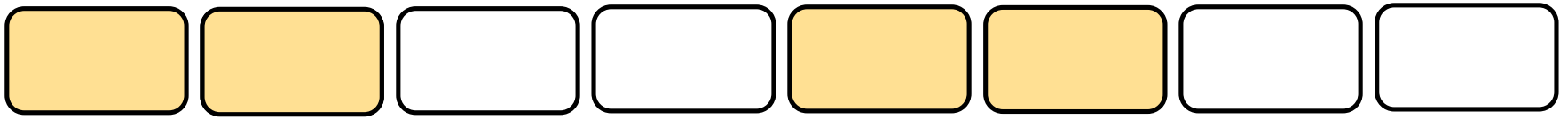
Notes:

- Commit state of a chunk has nothing to do with whether chunk is free or not.
  - A chunk which is in use will commit on demand when allocation happens, if needed.
- A chunk can contain a mixture of committed and uncommitted sections.
- Chunks and granules are aligned such that no chunk struddles a granule border without containing the whole granule.

Allocation – committing on demand



chunk

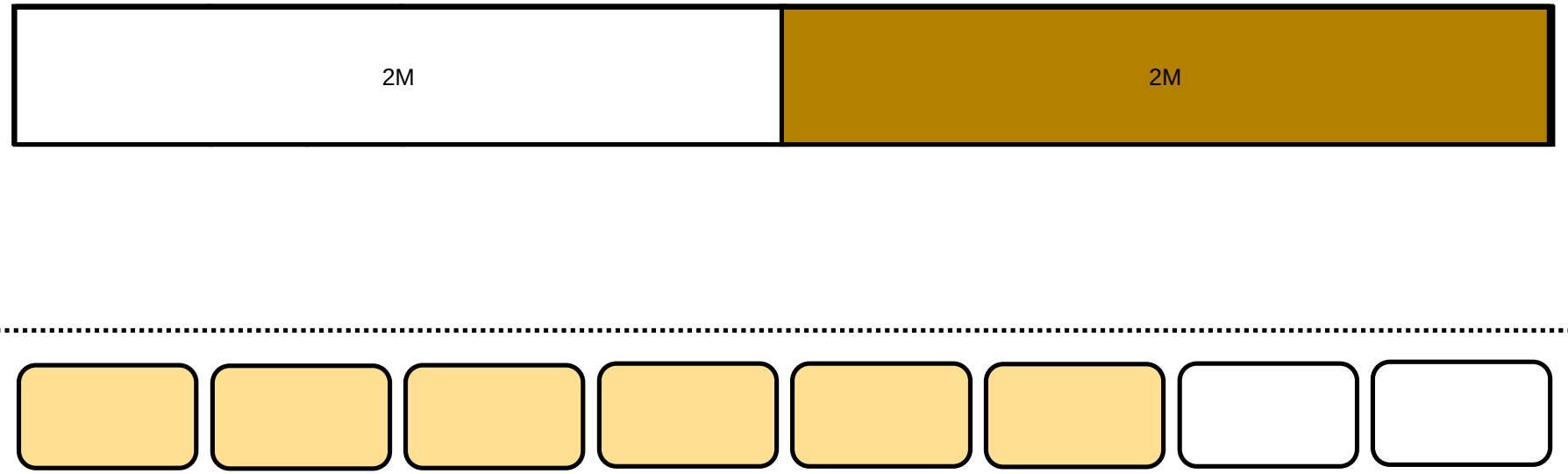


granules

Notes:

- Attempting to commit already committed range is a cheap noop.
- If committing hits a limit (MaxMetaspaceSize), the metaspace allocation fails (->GC and perhaps OOM)
- The commit mask bitmap inside VirtualSpaceNode is the “one truth” but we keep a per-chunk committed counter to speed up things
- A newly allocated chunk will start out with a committed range at the begin (configurable)

Freeing and uncommitting



Notes:

- All highly configurable:
  - how large a chunk has to be to be uncommitted on free;
  - uncommit on free or only after Metaspace GC purge operation;

Benefits of commit granules

A simple straightforward way to manage “checkered” committed regions.

Cheap to implement (bitmap) and to manage.

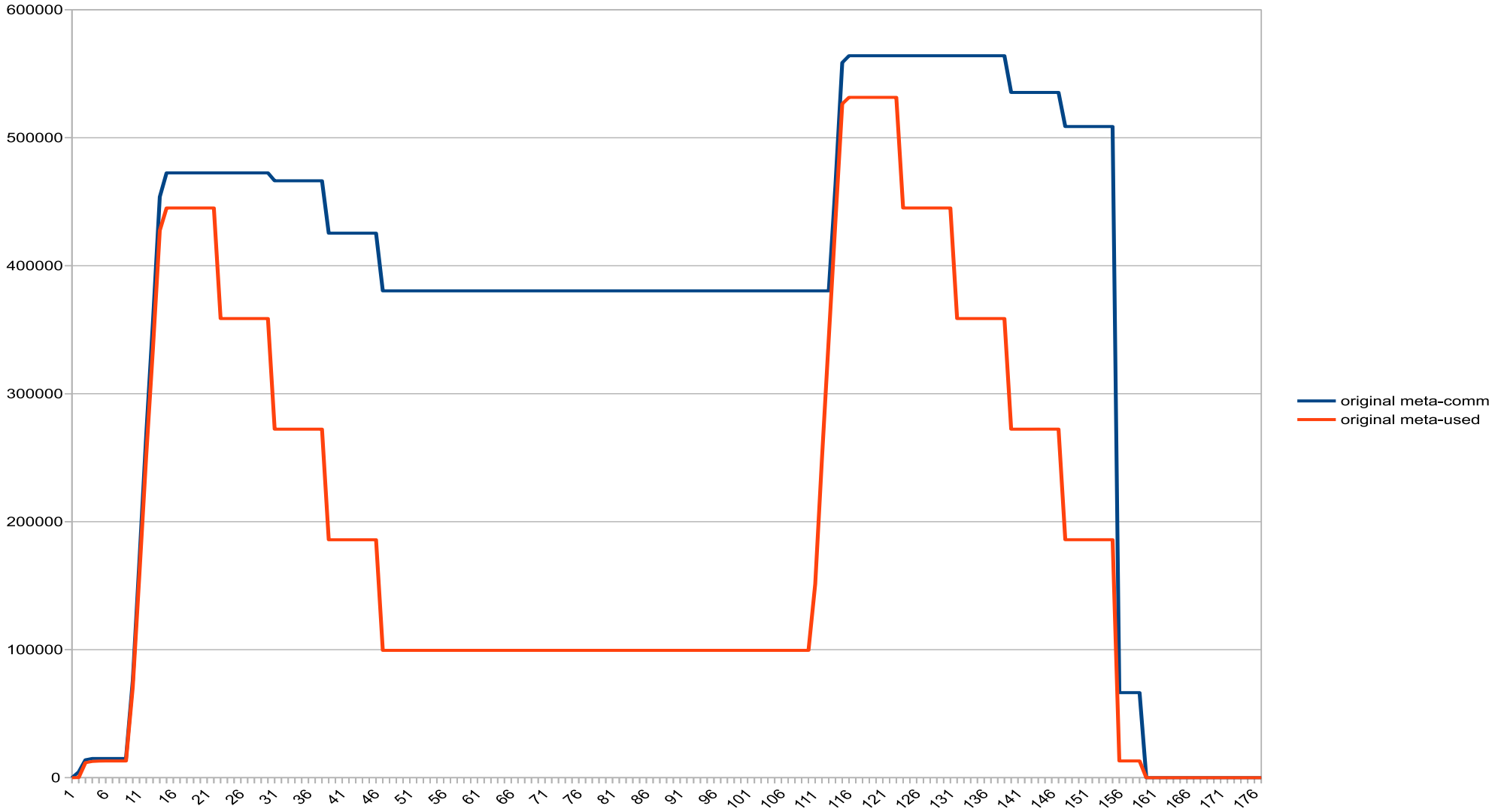
Easy to configure (commit granule size)

Independent on the upper layer – in this case the buddy style allocated chunks. Very few dependencies between those two layers.

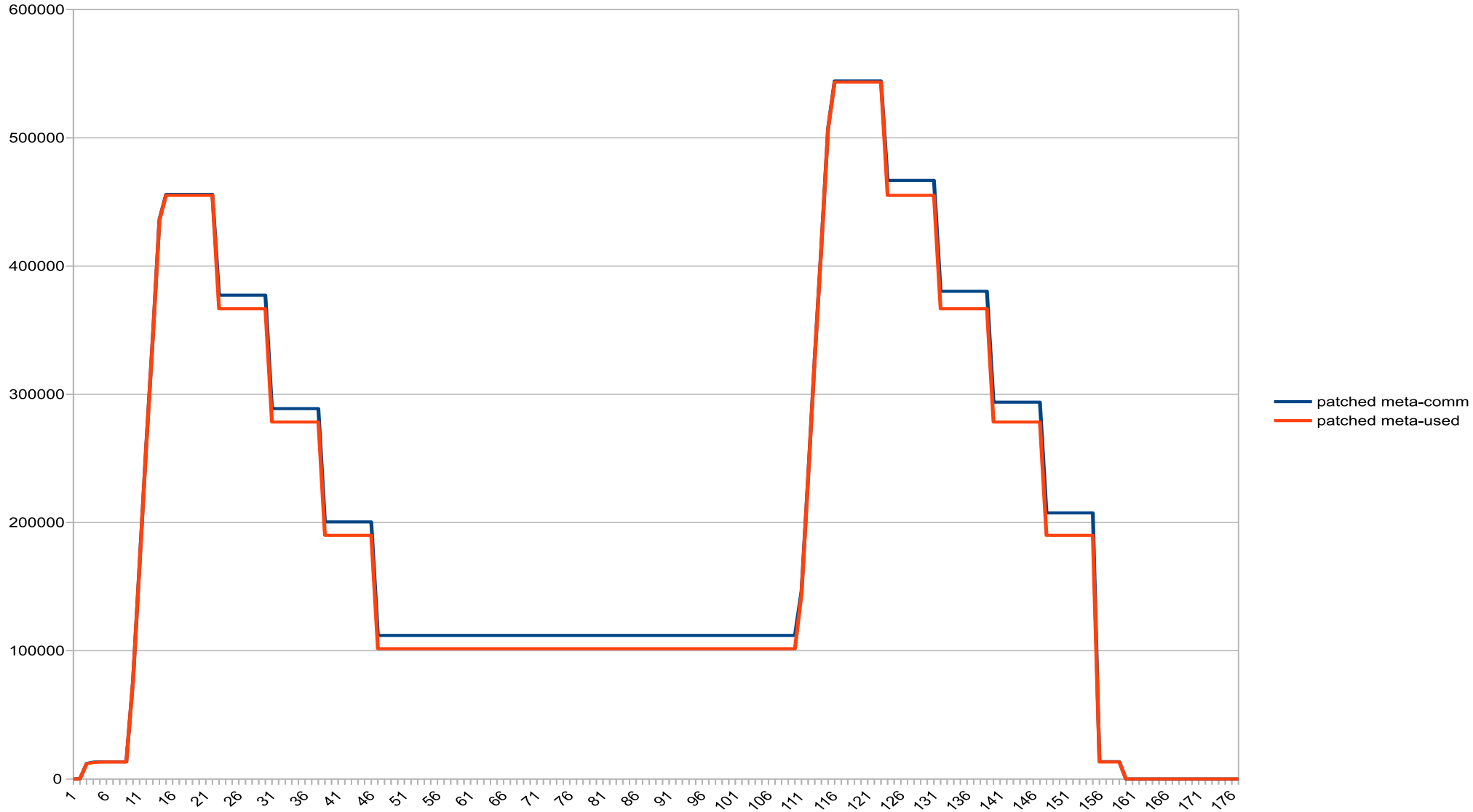
-> simple and robust

Results

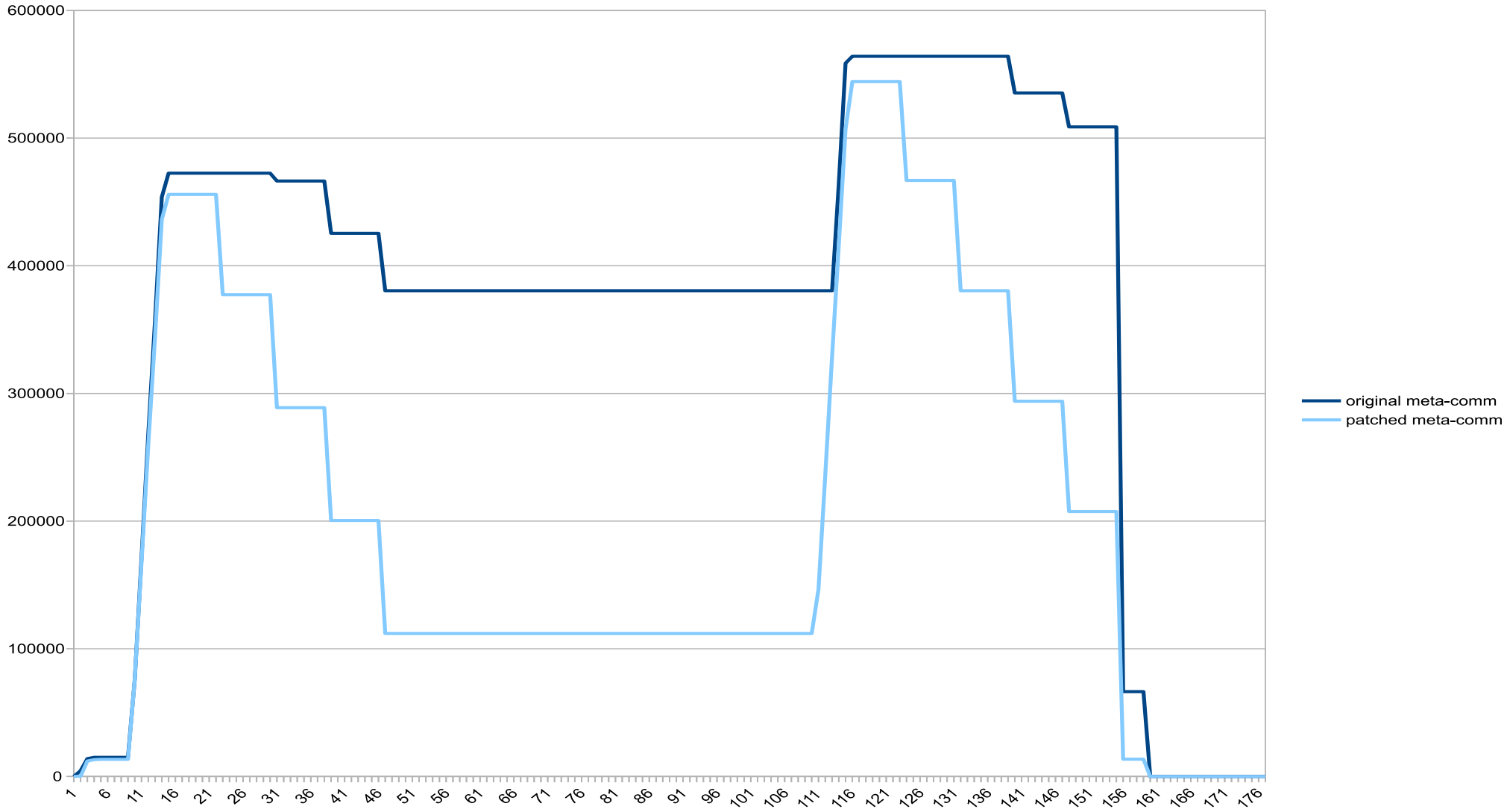
Parallel Loaders Test, MS committed vs used, Before



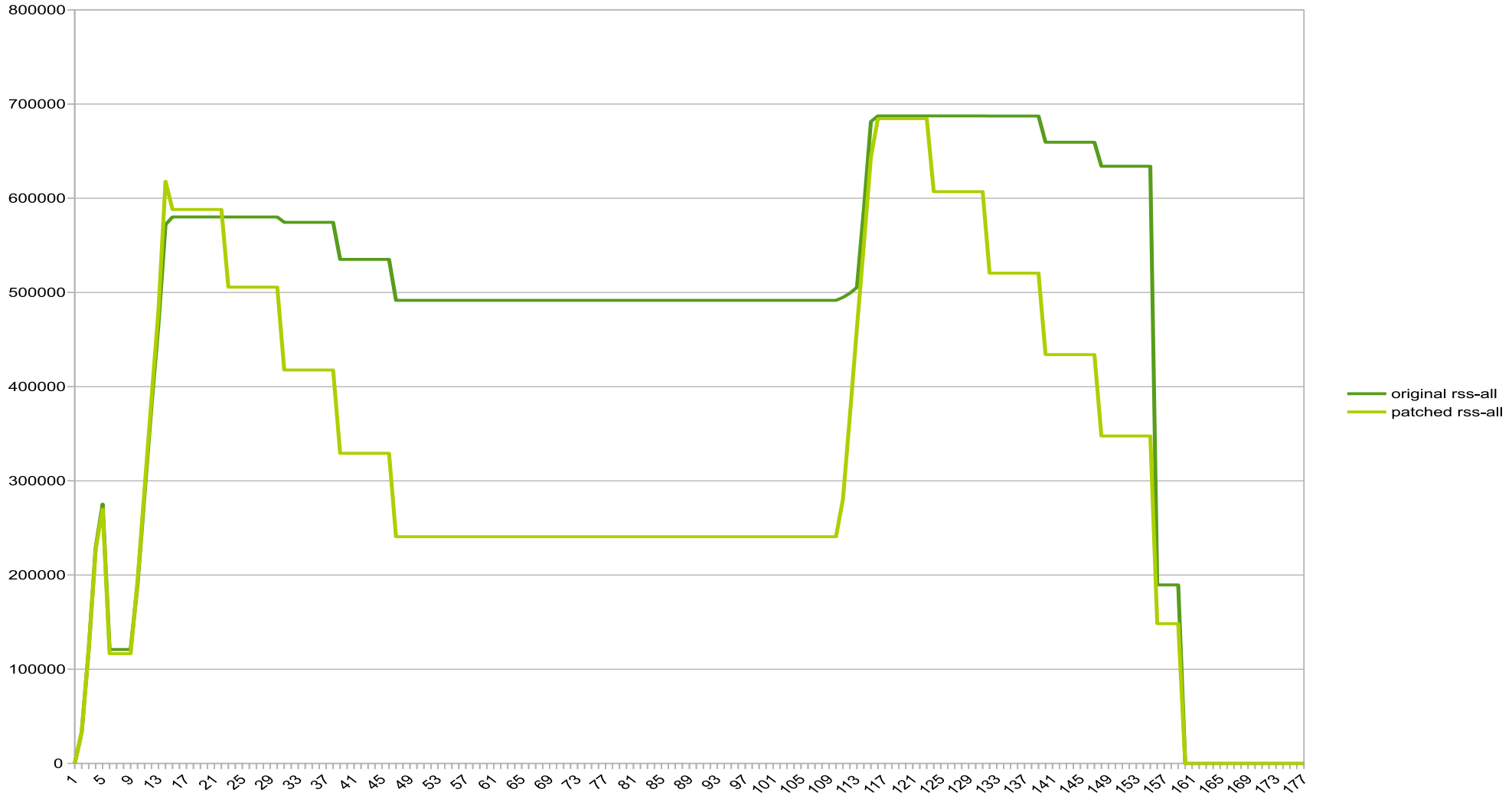
Parallel Loaders Test, MS committed vs used, After



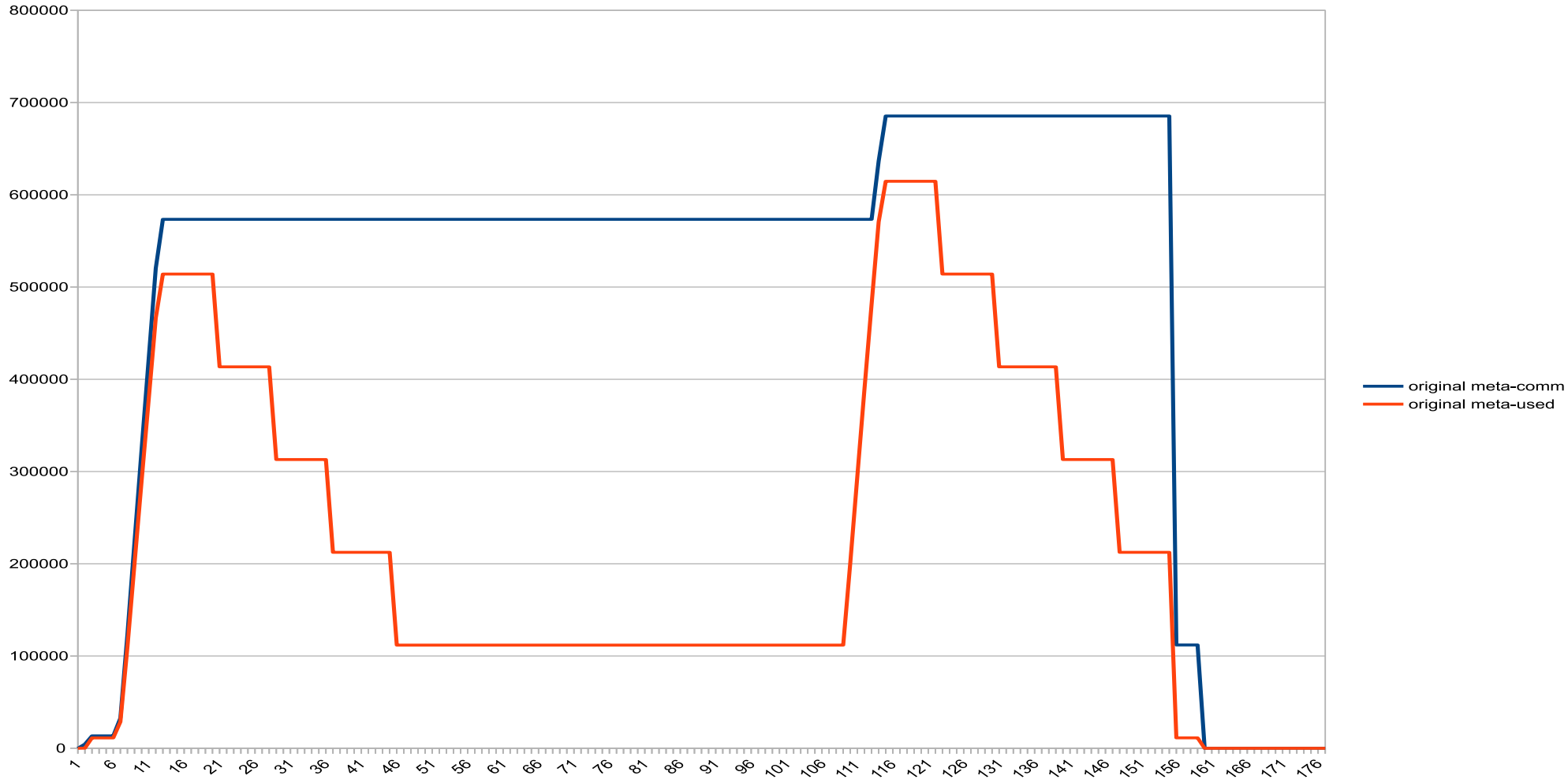
Parallel Loaders Test, MS committed, before vs after



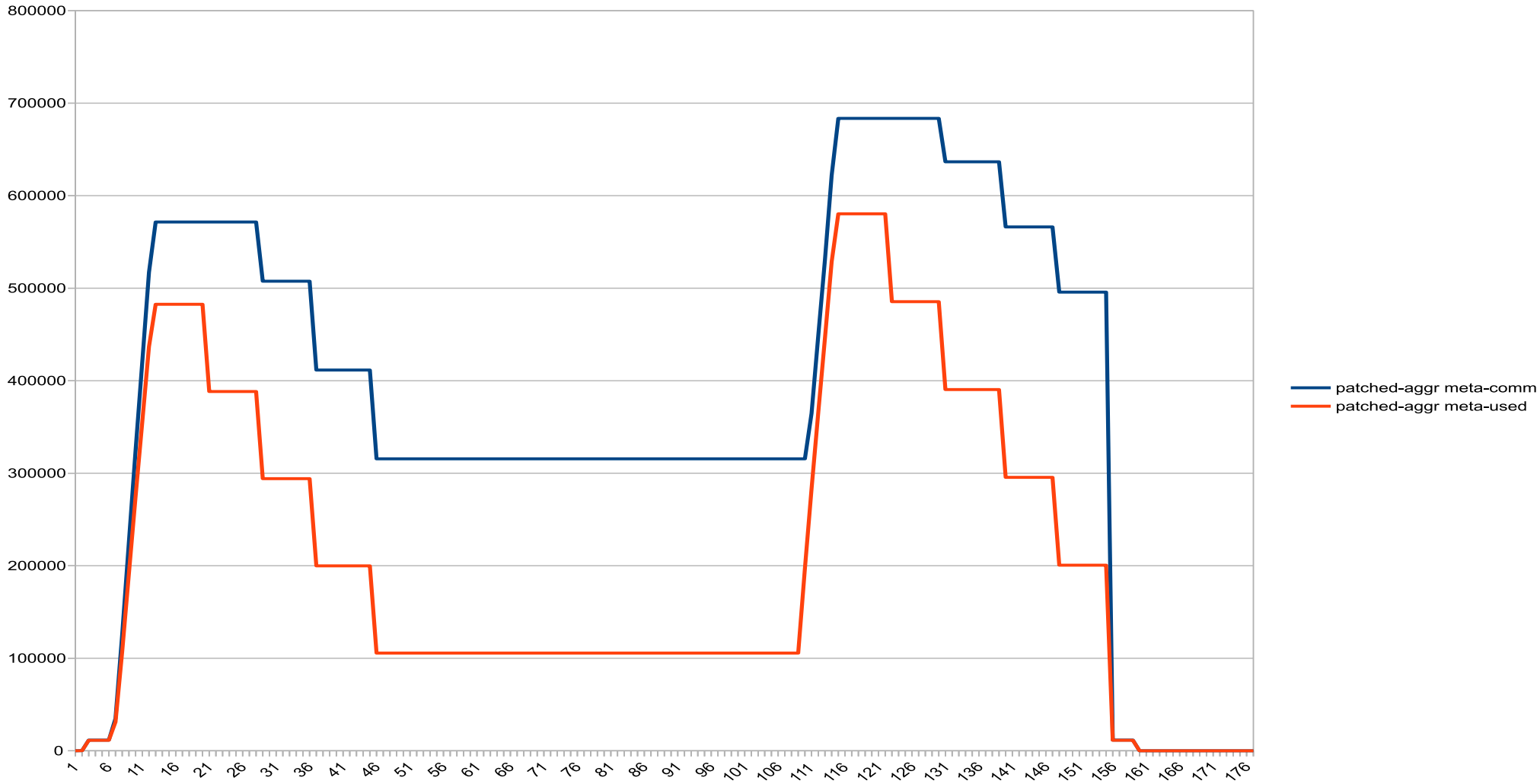
Parallel Loaders Test, Rss, before vs after



80000 micro loaders, MS committed vs used, before



80000 micro loaders, after, MetaspaceReclaimStrategy=aggressive



Reduced waste even without class unloading:

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

Switches

- MetaspaceReclaimStrategy=(none|balanced|aggressive)
  - “none” is for testing. Like old VM (purging is still done)
  - “balanced” is standard.
    - Commit granule size is 64K
  - “aggressive”
    - commit granule size 16K
    - Increases vm fragmentation but good results with micro loaders

- -XX:+MetaspaceSeparateMicroCLDs

Code is way cleaner now!

- Much less class/non-class code paths now. Not much code “knows” about class space at all.
  – E.g. chunk sizes are generic, not space specific

- Better sectioned. Sub parts are much easier to test in isolation.

- Loads of new gtests!

Show me the code

- http://hg.openjdk.java.net/jdk/sandbox/
  - branch "stuefe-new-metaspace-branch“
- Patch is about 20kloc
- Initial patch: http://hg.openjdk.java.net/jdk/sandbox/rev/bdf136b8ae0e
- Add Vitals via vitals patch at http://cr.openjdk.java.net/~stuefe/webrevs/metaspace-improvement/vitals.patch

Status 2019-10-09

- Allocator is stable enough for daily use (under CDT)
- Works with CDS, AppCDS
- Some small issues remain (e.g. Large pages in metaspace)
- No performance regressions found
- Still to do:
  – Fix issues
  – Improve comments
  – Contribute

I wish I could also…

- Make allocator better for one-class-loading-loaders (“micro clds”) – e.g. Reflection Delegators and Method Handle CLDs

- Still somewhat bad intrachunk waste (we always pay 1K for an InstanceKlass which is typically ~600byte)
  – Similar waste in non-class area

- Band aid:
  – -XX:+MetaspaceSeparateMicroCLDs
  – Moves all allocations from known Micro-CLDs into class space
  – Pro: we pay ultimately less since we are not split into class chunk and non-class chunk(s)
  – Con: Counts toward CompressedClassSpaceSize

- Idea: an own allocator for Micro CLDs
  – At least for the InstanceKlass structures in class space
  – Needs more thinking… but not impossible.

Future

- JDK 15 (14 probably too late)?
- JEP or as a simple RFE?
- SapMachine 11?
- SAPJVM 8?

- Its all very foggy right now. But I hope to at least be able to show something at Fosdem20. Lets see…

Questions?

Thank you.

Contact information:

Thomas Stüfe, SAP

THE BEST RUN SAP®