

The slab allocator sheaves post-mortem

Vlastimil Babka, SUSE Labs
Linux Plumbers Conference 2026

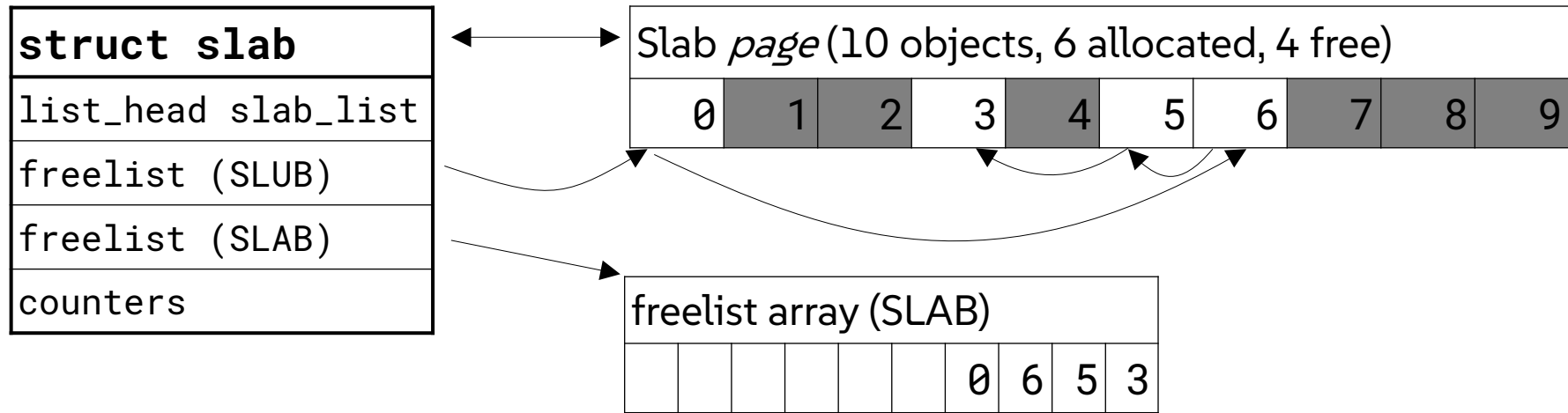
SLUB percpu sheaves – initial motivation (2023/2024)

- What's the next step after removing SLOB and SLAB, leaving only SLUB?
- Overall goal: remove various caching layers on top of the slab allocator
 - Usually arrays of object pointers
 - Some seem to do it just for performance reasons ([block layer](#))
 - BPF: allocate/free in restricted contexts (e.g. NMI handler) not supported by slab
 - Maple tree nodes: preallocate before an operation that can't fail in the middle, return unused objects afterwards
- Why is this bad?
 - Code duplication, maintenance overhead, wasted memory, in some cases cpu overhead
 - Cached objects out of allocator's control, cannot be consolidated or reused

SLUB percpu sheaves – initial motivation (2023/2024)

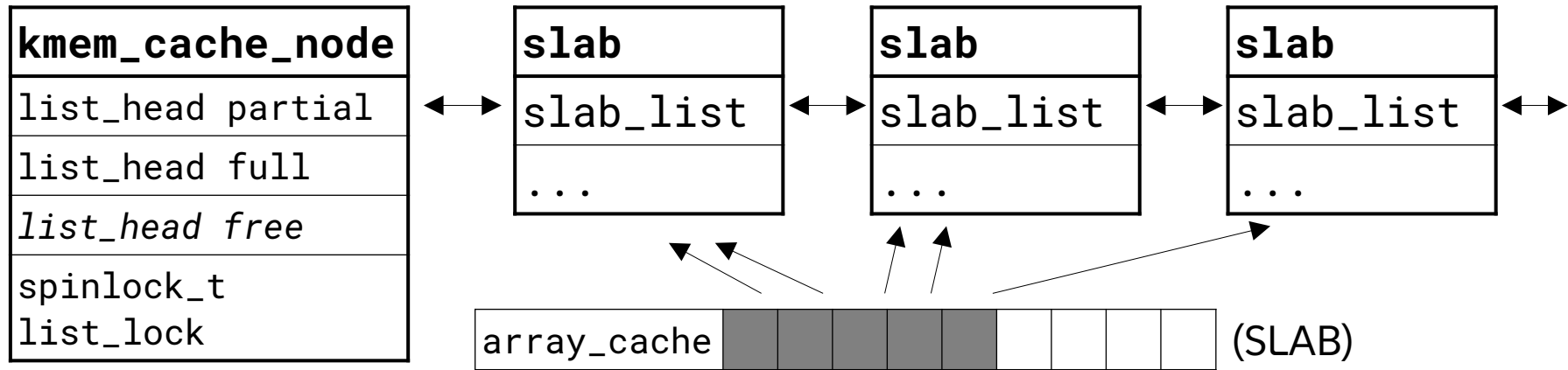
- Idea: once slab itself does what users need, caching on top is unnecessary
 - Build the necessary caching into the allocator itself
 - An array of object pointers in some form, but under the allocator's control
- Also try to improve performance for all slab users
 - Perhaps even simplify the implementation
 - Combine the good parts of SLUB and SLAB design?

SLUB (vs SLAB) basic functionality – single slab



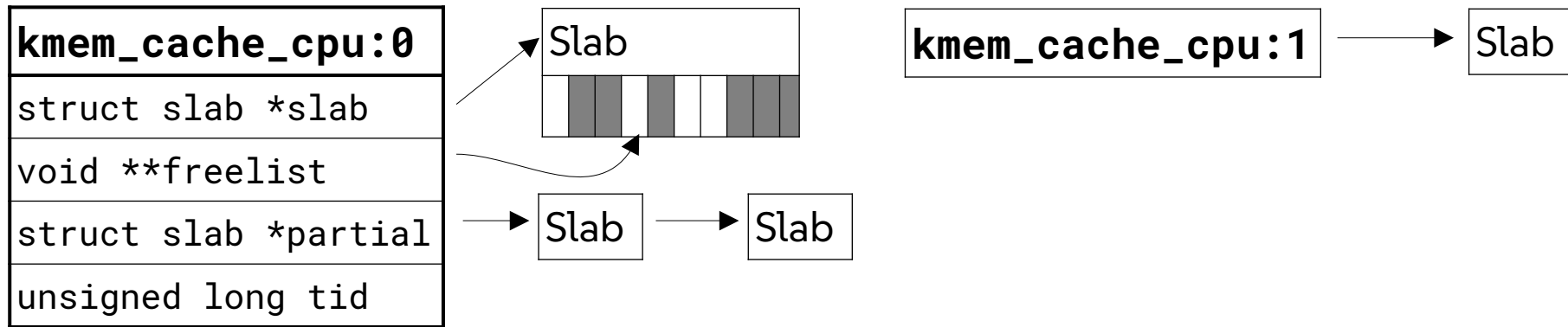
- SLUB has freelist embedded in free objects – single-linked, NULL-terminated
 - Allows allocating all objects in a single operation, multiple freelists in a single slab
 - Allows freeing to the slab with an atomic update of `freelist+counters` by the `try_cmpxchg128( )` operation

Multiple slabs, SLAB percpu caching



- List of slabs per NUMA node with a spinlock – would not scale, needs some form of percpu caching
 - Turns out the caching is much easier to implement for allocating than for freeing
- SLAB - array caches: percpu (only local objects), shared (same node), alien (freeing remote objects, needed periodic flushing) – helps NUMA locality
 - Part of the array flushed to slabs (or shared array) when full, part refilled when empty

SLUB percpu caching (before sheaves)



- Allocations – fast from the percpu slab (if available and from a matching NUMA node)
 - `this_cpu_try_cmpxchg128()` of `freelist+tid` to avoid irq or preempt disable
- Freeing – fastpath similar to allocation, but only if object is from the exact cpu slab
 - Freeing to different slab is much more likely than freeing to remote NUMA node in SLAB
 - Slowpath is `try_cmpxchg128()` (atomic), sometimes also spinlock for the partial list

SLUB percpu sheaves – basics (v8) (LWN)

- Same as [magazines](#) by [Bonwick](#), but wanted to avoid loaded [sic!] terms
 - Thanks to Matthew Wilcox for the terminology
- Sheaf: fixed-capacity (per `kmem_cache`) array of object pointers from slabs
 - Each cpu has two - `main` and `spare` sheaves, so that flushing or refilling is not partial
 - Per-node `barn` is a victim cache for empty and full sheaves (up to 10)
- Issues with SLAB design avoided by bypassing sheaves when freeing remote objects
 - A quick tracing session suggested it's only done in ~5% of the cases
 - No `alien` arrays that would need periodic flushing
- Percpu sheaves locking overhead limited by the `local_trylock_t` primitive
 - Uses cheaper `preempt_disable` instead of `irq_disable` in exchange for (rare) `trylock` failures

SLUB percpu sheaves – handing out prefilled sheaves

- Support a usage pattern where a number of objects might be needed in a critical section that cannot block (and thus reclaim memory to allocate) but also can't handle allocation failures
 - Maple tree store – exact number of (node) allocations not known beforehand, but the worst case scenario can be pre-calculated
 - Previously handled by preallocating for the worst case, freeing back most of the preallocated objects afterwards – ineffective even with bulk alloc/free
- `kmem_cache_prefill_sheaf(size, gfp)` - ideally removes the spare sheaf, refills it only when it has less than `size` objects, returns it to the caller
- `kmem_cache_alloc_from_sheaf()` - guaranteed to succeed until depleted
- `kmem_cache_return_sheaf()` - ideally restores the sheaf as spare as-is, avoids flushing if possible

SLUB percpu sheaves – kfree_rcu() batching

- The existing `kfree_rcu( )` implementation frees objects back to their slabs after the grace period
 - Batching exists to make it more efficient than a straightforward implementation
- Problem: freeing to percpu sheaves of many accumulated objects was overloading them
 - With `rcu_nocbs`, this freeing was restricted to some cpus only, others not refilled
- Solution: each cpu has additional `rcu_free` sheaf
 - `kfree_rcu( )` adds to this sheaf; when full, `call_rcu( )` called on the full sheaf
 - `rcu_free_sheaf( )` callback will try to put the sheaf to barn for reuse

SLUB sheaves timeline to mainline – initial merge

- Sheaves were merged in 6.18 (a LTS kernel...)
- Initially opt-in, enabled only for VMA's and maple tree nodes
 - Capacity given when creating the `kmem_cache`
 - Both utilize `kfree_rcu()`
 - Maple nodes also use the prefilled sheaves API
- The immediate result: SLAB fixes PRs for rc1 through rc7 (except rc4) ?!? (normally 1-2 per release)
 - Vast majority actually related to `kmalloc_nolock()`, not sheaves
 - One [memory leak fix](#) (logic error) for rc6

SLUB sheaves timeline to mainline - enabling for all caches

- Being opt-in is far from ideal though!
 - It was OK as the first step to validate the implementation, fix bugs...
- Non-trivial implementation and maintenance overhead for only few users
 - Still backed by another caching layer with arguably even less trivial implementation
- How to determine which caches should benefit from the opt-in?
 - What sheaf size? Should the size be fixed? Automatically selected?
- Thus, enabling sheaves for all caches was the next step
 - And then removing the existing caching based on cpu (partial) slabs

SLUB sheaves timeline to mainline - enabling for all caches

- The series ([v4, 22 patches](#)) was merged in 7.0
- Some fun with bootstrapping and recursion prevention
 - Sheaves are allocated by `kmalloc()`, initial percpu sheaves have to be instantiated at cache creation, so how to create caches for `kmalloc` without allocating sheaves?
 - Allocating from a `kmalloc()` cache might result in allocating a sheaf, which can (depending on the size) be actually allocated from the very same cache?
 - Both solvable and not completely new problems in the history of slab allocators
- Sheaf capacity calculated automatically to roughly match how many objects the cpu (partial) slabs would cache on average - subject to possible future tuning
 - Also rounded up to nearest `kmalloc()` object size
 - Any explicitly specified size now treated as a minimum

SLUB sheaves timeline to mainline - enabling for all caches

- Lots of code removed afterwards :) including usage of `this_cpu_try_cmpxchg128()`
 - `try_cmpxchg128()` usage remains in the slow paths for slab's freelist+counters
- Lots of fun with a memory leak due to missing "break ;" in an earlier version
 - Not found by Chris Mason's LLM powered review until retrospectively told to look for leaks
 - After his prompts adjustments, now should be found in [5 of 6 cases](#)
- For 7.0, pull requests with fixes were needed for rc1, rc2, rc3
 - All had some fixes related to the sheaves enablement
 - Spurious allocation failed [warning](#), memalloc profiling [warning](#), mempool perf [regression](#), yet another (corner case, by code inspection) [memory leak](#)

So what about performance?

- The initial implementation was validated on [in-kernel microbenchmarks](#)
 - Verify the fast paths are faster thanks to `local_trylock_t`, more often hit when freeing
- Due to being such a low-level implementation detail, it appears difficult to benchmark reliably
 - Things like subtly different memory layout may play a role due to processor caches
 - Hitting the slow paths might affect workloads differently depending on their alloc/free patterns
- Suren Baghdasaryan had [measured](#) `mmap()` operation improvements
- For enabling sheaves on all caches, there were benchmarks done by Harry Yoo and Petr Tesařík
 - In general showing an improvement, but with specific cases showing worse performance
 - But that was with older versions of the patches, e.g. with the memory leak still present
- Many performance aspects are the matter of tuning – sheaf sizes, barn sizes, and various heuristics

What are the underlying tradeoffs?

- Thanks to [Christoph Lameter](#) for summarizing them
- CPU slabs (before sheaves): consecutive allocations served from the same CPU slab
 - Spatial locality, good for TLB overhead
 - Worse for cache performance (objects more likely to be cold)
- Sheaves: allocations are typically returning recently freed objects (LIFO)
 - Temporal locality, good for cache performance (objects more likely still hot)
 - Potentially scattered from many slabs (worse for TLB)
 - Might also increase memory overhead (but nobody reported that as an issue so far)
- Object freeing – fastpath is more likely with sheaves, and a bit faster thanks to `local_trylock_t`
 - Slowpath is identical – `try_cmpxchg128()`

Some reported regressions and improvements

- [Eric Dumazet](#): “tcp_r r with 10,000 flows goes from 25 Mpps to 40 Mpps on AMD Turin.”
 - “lock cmpxch16b latency can be over 30,000 cycles”
- 20% will-it-scale [brk1 regression](#) (and [mmap2](#)) with maple node caches enabling sheaves in 6.18
 - Middle of series commit where the usage of prefilled sheaves comes later
- 57% will-it-scale mmap2 [improvement](#) thanks to a small [corner case tweak](#) in 7.0
- 7% fork+execve [regression](#) after enabling sheaves for all caches (7.0)
 - The caches involved (vma, maple nodes) were already converted to sheaves in 6.18
 - But the original cpu (partial) slabs layer remained (a mistake in hindsight)
 - Effectively ~doubling the percpu caching capacity
 - Some workloads would benefit from the extra capacity in 6.18 and lose it again in 7.0
- 15% [xfs lockofd improvement](#) after enabling sheaves for all caches
 - Could be similar case, cpu (partial) slabs removed by later commits

Sheaves development after 7.0

- [Fix](#) for performance regressions on memoryless nodes
- Optimizations for [sheaf refill](#) with [5.4% brkl improvement](#)
 - Oops, tried to avoid spinlock too much, caused a rare but nasty [ABA bug](#)
 - Oops, the fix is not for free, 2.3% stress-ng [pagemove regression](#)
- Another [sheaf refill optimization](#) with [7% mmapl](#) and [8% pagemove](#) improvements
- Yet another memory leak in bulk freeing (currently not possible to hit in practice) [fix](#)
 - Finally [refactor](#) the damn [thing](#)
- Various cleanups, usually as a followup to fixes

The most recent sheaves development

- Started with Hao Li observing a `list_lock` contention on maple node cache
 - Proposed [fix](#) seemed to help but seemed like a workaround
 - Further [investigation](#) found the culprit: `kmem_cache_prefill_sheaf()` skipping the barn
 - But that's the main allocation path for maple nodes
 - And the freeing side using `kfree_rcu()` found the barn full and had to flush
- The [fix](#) improves `mmap1` throughput by 24%, validated with internal stats metrics:

metric	baseline	patched	change
alloc_slab	7,100,651	159,739	-97.75%
barn_get	849	194,460,392	+22904539.81%
barn_put	851	182,306,701	+21422544.07%
barn_put_fail	260,400,484	145,522,675	-44.12%
free_rcu_sheaf	8,332,839,104	10,490,535,227	+25.89%
free_slab	7,099,788	158,297	-97.77%
sheaf_flush	8,332,823,922	4,656,725,764	-44.12%
sheaf_prefill_fast	3,337,502,163	4,196,506,116	+25.74%
sheaf_refill	8,343,794,364	4,657,509,724	-44.18%

- Lesson: when introducing a caching layer, make sure it's used from all the relevant paths

Related and future work

- More suitable for `kmalloc_nolock()` - a variant for `alloc/free` in restricted context such as NMI
 - Replaces BPF object caches (one of the original motivation of sheaves)
 - Without being based on sheaves (but also merged in 6.18)
 - Some its complexity and limitations (mainly due to `PREEMPT_RT`) were removed with CPU slabs replaced by sheaves
 - But much complexity remains and we keep learning about it...
- Future work on sheaves
 - Integrate better with pre-existing `kfree_rcu()` batching
 - Now mostly bypassed
 - Tunable sheaf capacity, now exposed as read-only in `sysfs`
 - Better self-tuning as well?

Effects on code size - merging of sheaves in 6.18

15 files changed, 2280 insertions(+), 1458 deletions(-)

include/linux/local_lock_internal.h	9	+-
include/linux/maple_tree.h	6	+-
include/linux/slab.h	47	+++++
lib/maple_tree.c	667	+++++++-----

lib/test_maple_tree.c	137	-----
mm/slab.h	5	+
mm/slab_common.c	34	+++
mm/slub.c	1756	+++++++
+++++		
+++++		
mm/vma_init.c	1	+
tools/include/linux/slab.h	165	+++++++
tools/testing/radix-tree/maple.c	514	+++-----

tools/testing/shared/linux.c	120	+++++++---
tools/testing/shared/maple-shared.h	11	++
tools/testing/shared/maple-shim.c	7	+
tools/testing/vma/vma_internal.h	259	+++++++-----

mm/ only: 5 files changed, 1790 insertions(+), 53 deletions(-)

Effects on code size – replacing CPU slabs with sheaves

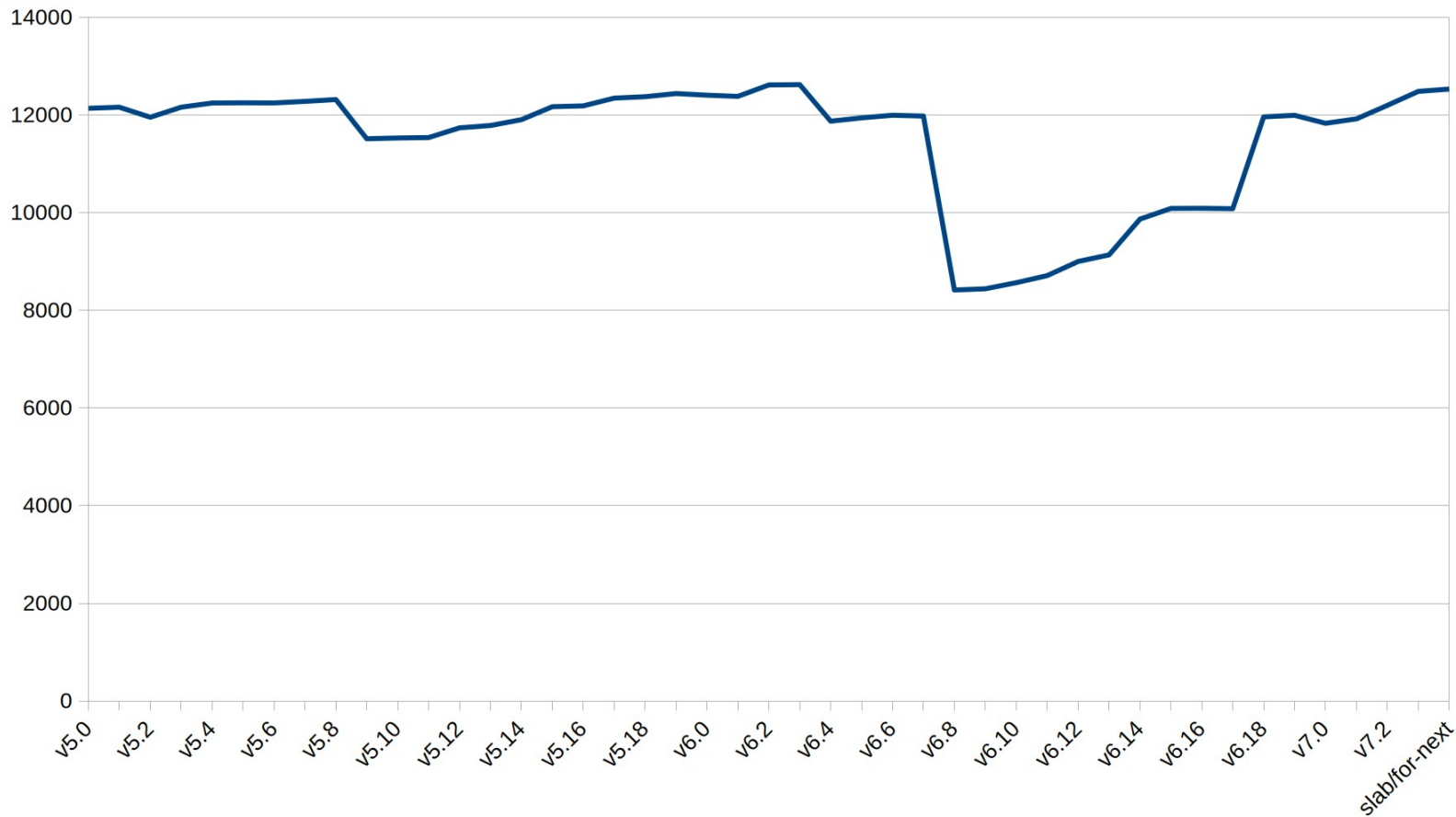
- The diffstat of slab/for-7.0/sheaves

```
- include/linux/slab.h |      6 -  
  mm/Kconfig          |     11 -  
  mm/internal.h       |      1 +  
  mm/page_alloc.c     |      5 +  
  mm/slab.h           |     65 ++---  
  mm/slab_common.c    |     86 +++++--  
  mm/slub.c           |    2736 ++++++  
+++++-----  
-----  
-----
```

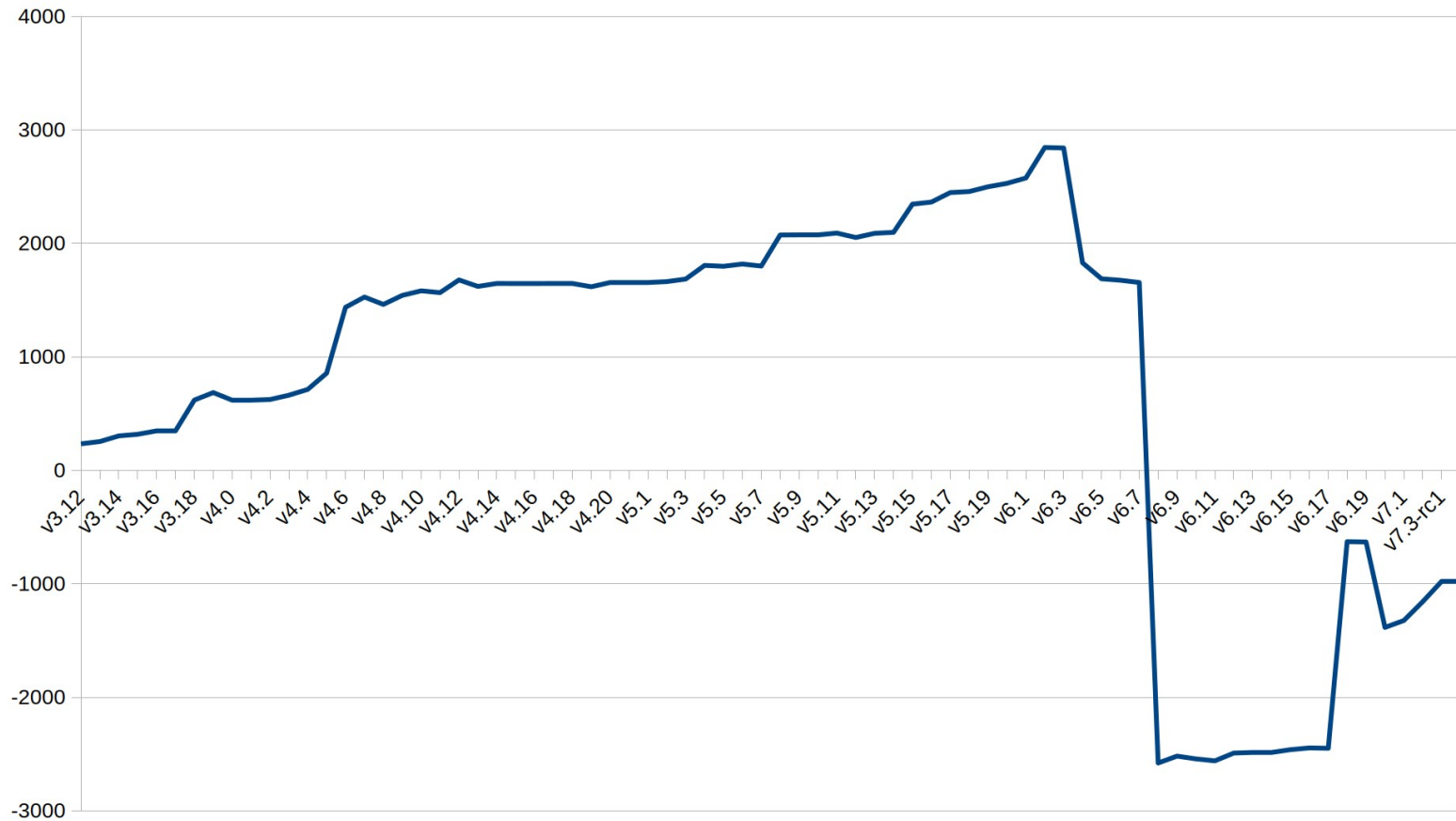
7 files changed, 1081 insertions(+), 1829 deletions(-)

- Together with sheaves introduction: net addition of 989 lines (mm only) or 74 lines (all)

Evolution of slab allocator(s) code size

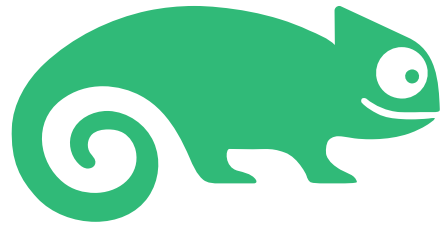


Evolution of personal diffstat (lines added minus removed)



Big Thanks To...

- Harry Yoo, Hao Li, Liam Howlett, Suren Baghdasaryan, Matthew Wilcox, Paul McKenney, Uladzislau Rezeki, Alexei Starovoitov, Shengming Hu, Hao Ge, Petr Tesarik, Pedro Falcato, Mateusz Guzik, Christoph Lameter, Sebastian Andrzej Siewior, Qing Wang, Tytus Rogalewski, Ming Lei, Hyunwoo Kim, Eric Dumazet, Aishwarya Rambhadrn, kernel test robot folks
- Sorry if I forgot anyone!



SUSE

Thank you

Questions?