Moving swapping infrastructure to Rust

Vitaly Wool, Konsulko

Kangrejos 2025

Swapping (paging)

- using secondary storage to store and retrieve data
  - secondary storage is usually an SSD or flash device
  - saves memory by pushing rarely used pages out
- trade memory for performance?
  - reading and writing pages may be quite slow
- use RAM to cache swapped-out pages
  - compress swapped-out pages, or there's no gain
- trade performance for memory?
  - in some sense, but we also get more flexibility

zswap: compressed write-back cache

- compresses swapped-out pages and moves them into a pool
  - when the pool is full enough, pushes the compressed pages to the secondary storage
  - pages are read back directly from the storage when needed
- compression is implemented using crypto API
  - several compression backends (lz4, lzo, zstd, gzip...)
- allocation is implemented using zpool API
  - Was: 3 allocation backends (zbud, zsmalloc, z3fold)
  - Now: only zsmalloc remains
  - An attempt to remove zpool has been made

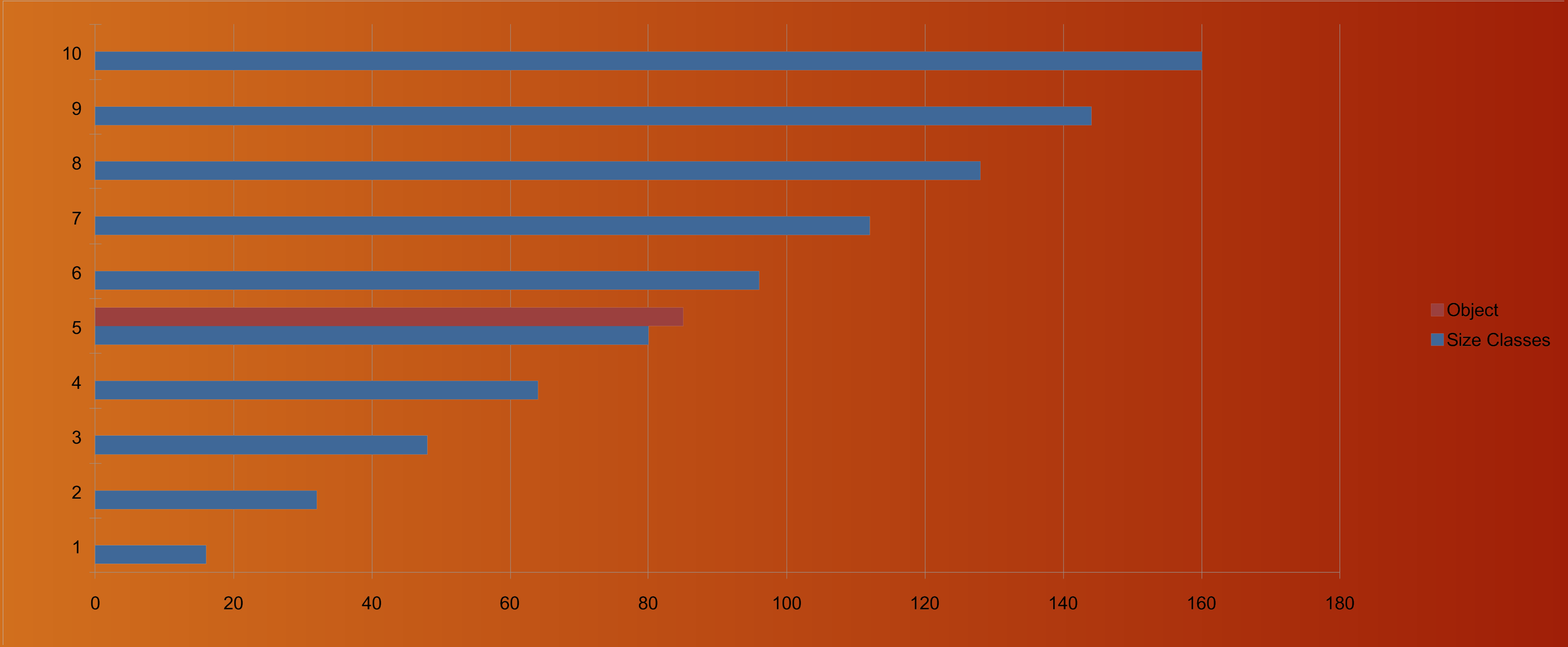
ZRAM: ramdisk with compression

- RAM block device driver with on-the fly compression
  - Basically a ramdisk on steroids
- Selectable compression backend
  - Selectable via the crypto API
  - LZO, LZ4, zstd etc.
- Non-selectable allocation backend
  - Always zsmalloc via zsmalloc’s own API
- Fully featured block device
  - Mostly used as a swap storage in Android
  - Zswap is not used in this configuration

zsmalloc

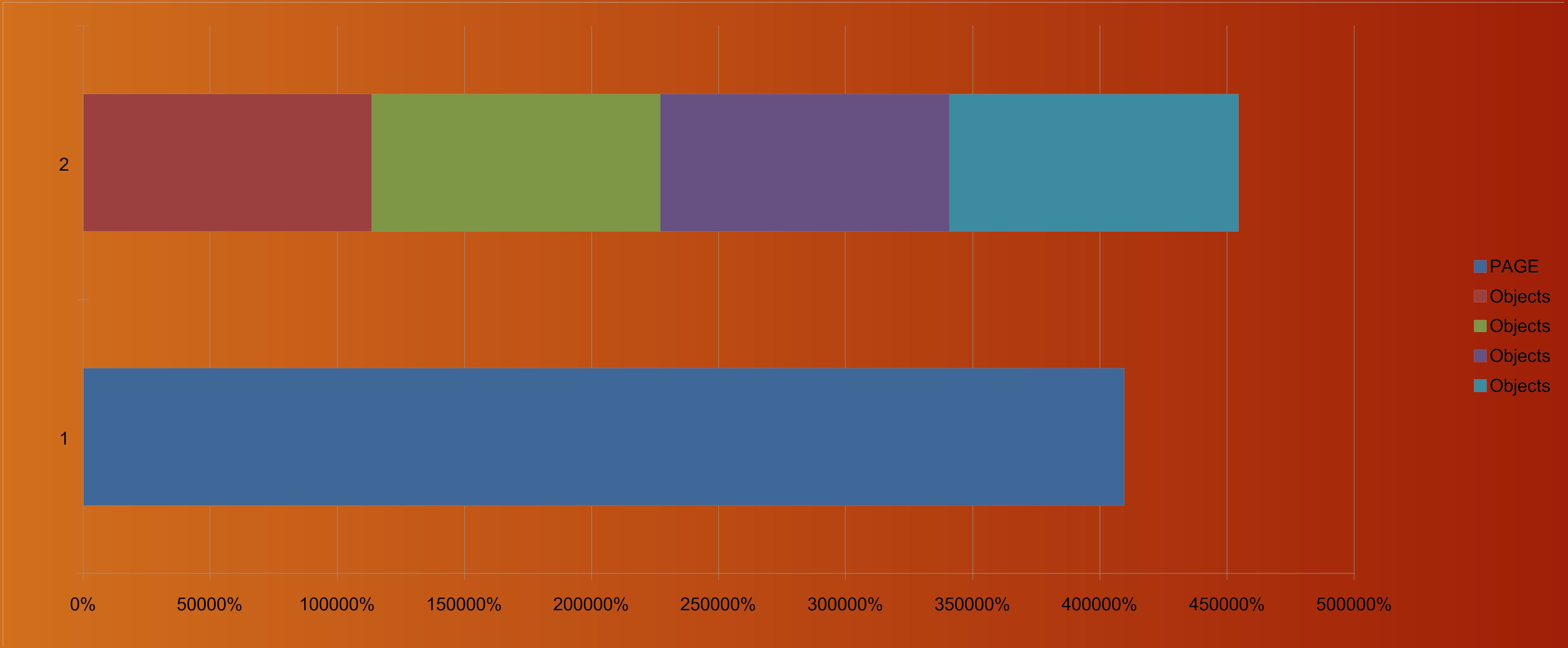
- Mature allocator backend
  - Provides very good compression density
  - Fast and scalable
- Rather complex implementation
  - Objects are divided into 255 size classes
  - objects of the same class stored consequently within a page
  - Some objects span across 2 pages
- Doesn’t work well in 16+K page setups
  - Lower granularity
  - Redundant data copies

zsmalloc



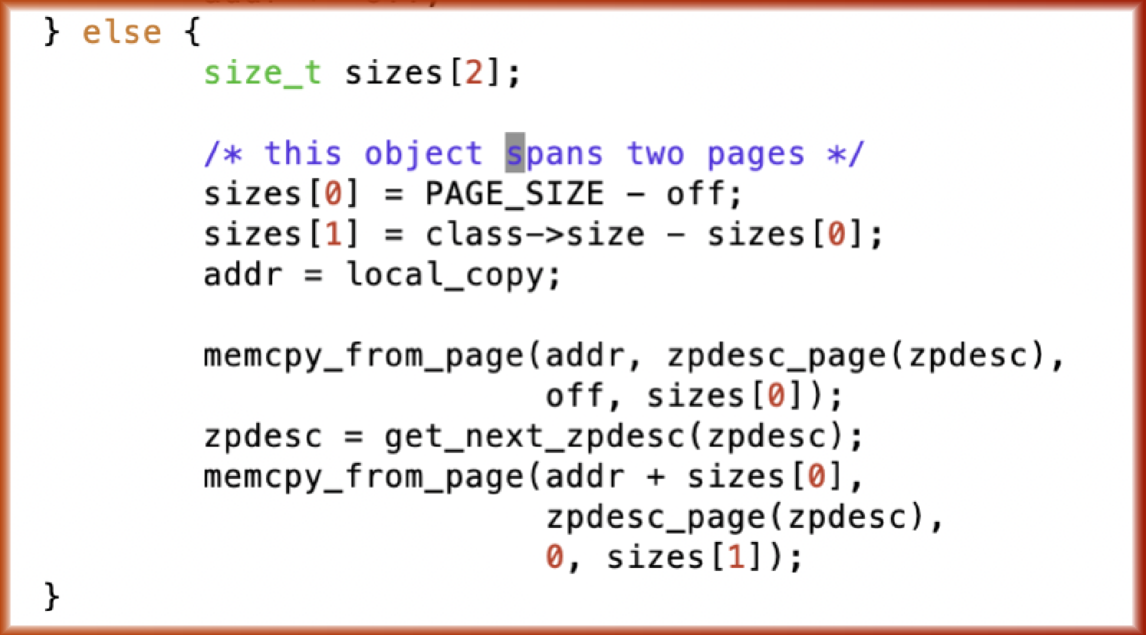
10
9
8
7
6
5
4
3
2
1
0
20
40
60
80
100
120
140
160
180
Object
Size Classes

zsmalloc’s spillover



zsmalloc’s spillover

- There’s nothing wrong about it
  - Still one has to do redundant copying
  - Compaction is required to keep fragmentation at bay
- There’s no easy way to prevent spillover
  - Class sizes are not 2^x multiples
  - “tail” bytes will be lost
- But we can organize page blocks
  - Minimize unused “tails”

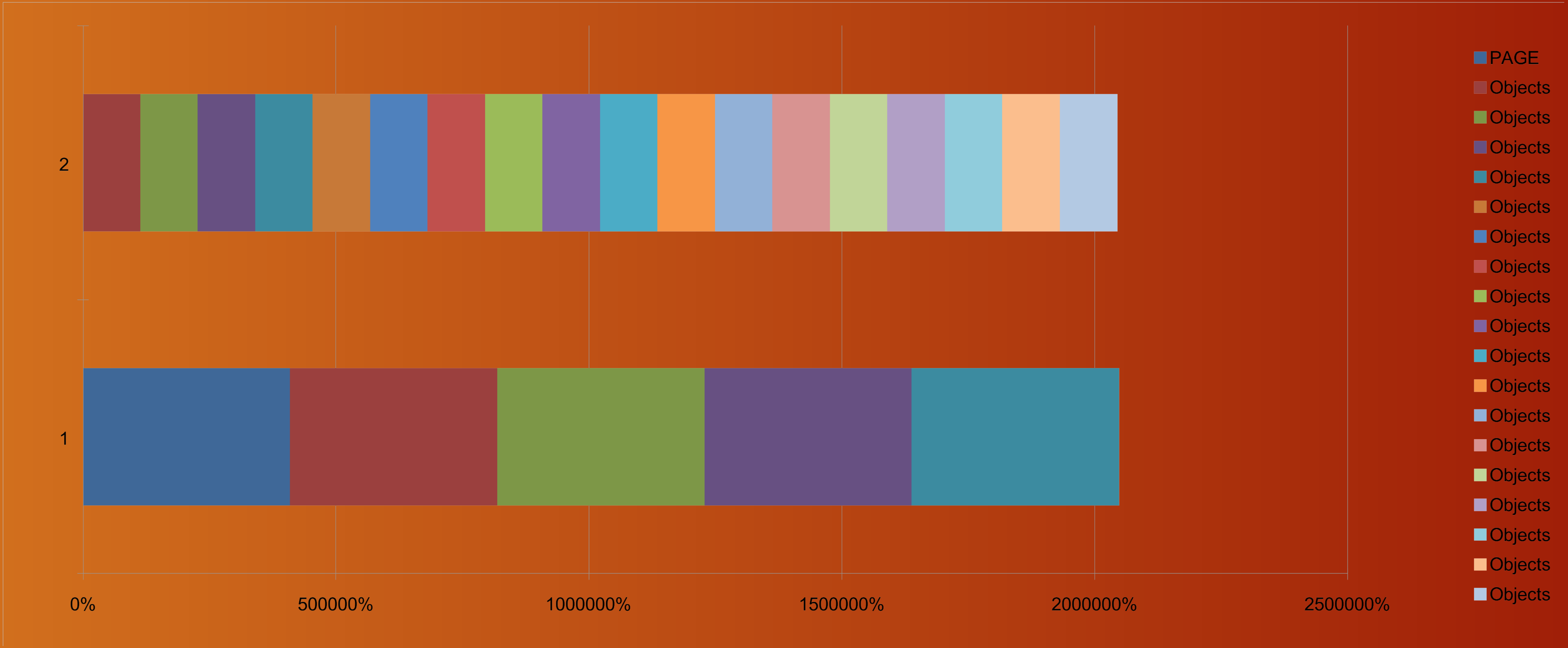


```
} else {
        size_t sizes[2];

        /* this object spans two pages */
        sizes[0] = PAGE_SIZE - off;
        sizes[1] = class->size - sizes[0];
        addr = local_copy;

        memcpy_from_page(addr, zpdesc_page(zpdesc),
                         off, sizes[0]);
        zpdesc = get_next_zpdesc(zpdesc);
        memcpy_from_page(addr + sizes[0],
                         zpdesc_page(zpdesc),
                         0, sizes[1]);
}
```

zblock

- Based on 2 simple ideas
  - With the recent advancements in vmalloc/vmap, one doesn’t need to reinvent the wheel
  - Divide large blocks into an array of same size objects
  - These same size objects (slots) don’t have to be of 2^x size
- Small code footprint and easy to understand concept
- How it relates to zsmalloc
  - 4K pages: comparable compression density, comparable performance
  - 16K pages: better compression density, better performance

zblock: no spillover



What we could move to Rust

- ZRAM was the first and the best candidate
  - It’s just a device driver
  - Lots of rarely used / unused code
  - Deployed almost exclusively in Android
  - Rust block device infrastructure wasn’t quite ready yet
- ZRAM_Rust will have to use zsmalloc C API
  - Reimplementing custom API in Rust?
- Moving zsmalloc to Rust is very complicated
  - Lots of code to rewrite
  - fiddling with low level mechanisms/data

What have we done?

- We noticed suboptimal zsmalloc performance with 16K pages
- The suggested way to go was:
  - Implement a new allocation backend in Rust
    - Well, in fact, reimplement zblock in Rust
  - Implement Rust zpool API
    - Patchset submitted but is on hold
  - Zblock.rs will communicate with zswap via that API
  - Keep C zswap implementation
  - Don’t do anything with ZRAM (yet)
- And that’s exactly what’s been done :)

Obstacles

- zpool API removal attempt
  - Still in mm-unstable, hopefully won’t get into 6.18
    - Motivation: “zpool is redundant, but you can add something aike to enable build time allocator choice”
    - Doesn’t sound like a good option for Rust
- Some MM stuff missing on the Rust side
  - Would gladly use vfree_atomic()
  - kmemcache
- Incomplete RCU implementation on the Rust side
  - Zblock would benefit
  - Toy implementation (out of tree)
  - Field projections (won’t be ready tomorrow)

Way forward (wishful thinking)

- Try to keep zpool API
- Proceed with the zpool API in Rust
- Cleanup and submit zblock.rs
- Cleanup and submit Rust ramdisk driver
- Extend the ramdisk driver to be a replacement for ZRAM
  - Think about Rust only API for zblock to be used by ZRAM_Rust
- Shmem in Rust?
  - Would be good for Rust DRM drivers
  - Too ambitious

That’s it, thanks for your attention

WHAT ELSE?