Decoupling ZRAM from a specific backend

Vitaly Wool, Konsulko Group

About me

Konsulko Group

- Linux since 1998
- Embedded Linux since 2003
- Worked for MontaVista
- Currently living in Sweden (Skåne)
- Staff Engineer at Konsulko Group
- Managing Director at Konsulko AB

About this presentation

Konsulko Group

- Swapping: ZRAM and zswap (frontends)
- Swappng: backends
- ZRAM over zpool
- Comparisons and charts
- Conclusions

Swapping:
ZRAM and zswap

Swapping and compression

Konsulko Group

- using secondary storage to store and retrieve data
  - secondary storage is usually a hard drive or a flash device
  - saves memory by pushing rarely used pages out
- trade memory for performance?
  - reading and writing pages may be quite slow
- How can we optimize that?
  - Cache some pages before actually writing them to a storage
  - That calls for compression, otherwise there's no win

zswap

Konsulko Group

- zswap is a *frontswap* implementation
  - i.e. it "intercepts" swapped-out pages
- zswap is a compressed write-back cache
  - compresses swapped-out pages and moves them into a pool
  - when the pool is full enough, pushes compressed pages to the secondary storage
  - pages are read back directly from the storage
- zswap is not self-sufficient
  - mostly targets desktop and server systems with "real" secondary storage for swap

ZRAM

Konsulko Group

- ZRAM is a block device with on-the-fly compression and decompression
  - Stored in RAM like an ordinary ramdisk
- Alternative to zswap primarily for embedded devices
  - no backing storage necessary
  - pages swapped to compressed RAM storage
- ZRAM is self-sufficient (can be used standalone)
  - Mostly targets embedded systems where secondary storage is limited or can't be used for swapping for other reasons

ZRAM and zswap

Konsulko Group

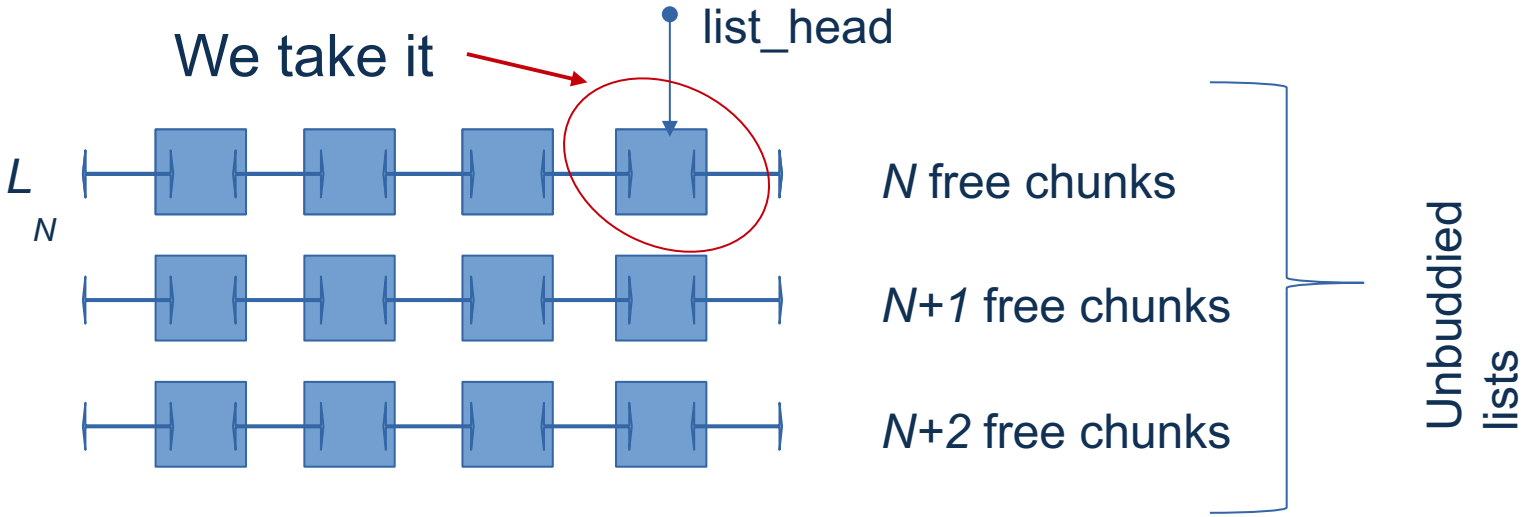
- Both may be considered swapping front-ends
  - Since both process swapped-out pages
  - And store them compressed in RAM
- Both aim for saving RAM space
  - Need to place compressed objects *tightly*
- Need a "backend" to store compressed pages
  - IOW, a compressed page allocator

Swapping: backends

zbud: the first backend (zswap)

Konsulko Group

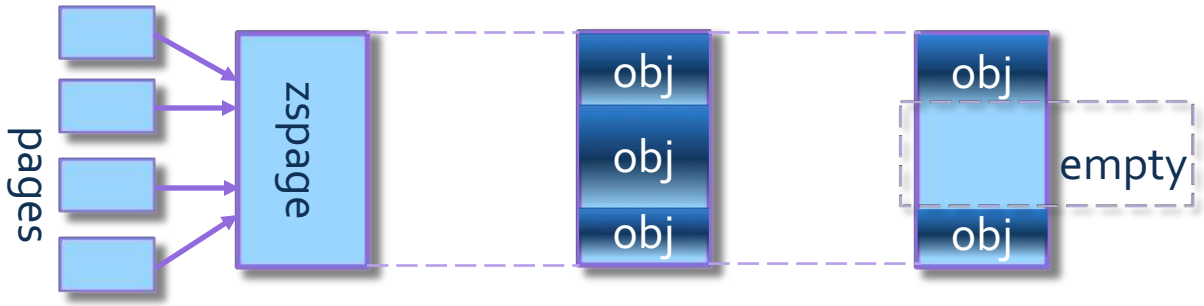
- stores up to 2 objects ("buddies") per page
  - One bound to the beginning, one bound to the end
  - Round object size to *N* chunks
  - Look for the first *unbuddied* from L_N (the list of all unbuddied objects with *N* free chunks)



zsmalloc (allocator for ZRAM)

Konsulko Group

- 2^N physical pages to form one *zspage*
- Compressed objects placed contiguously
  - Can span over 2 zspages
- Fragmentation issues have to be addressed



zsmalloc and zbud: applicability

Konsulko Group

ZRAM

- zsmalloc
  - high compression ratio
  - reclaim not used

zswap

- zbud
  - fast and simple
  - supports reclaim
- zsmalloc
  - high compression ratio

z3fold: new kid on the block

Konsulko Group

- spun off zbud
- 3 objects per page instead of 2
  - Higher compression than zbud while still simple
- ZRAM ready: can handle PAGE_SIZE allocations
  - zbud would have to be slightly modified to allow that
- work started after ELC 2016 in San Diego
  - In mainline kernel since 4.8

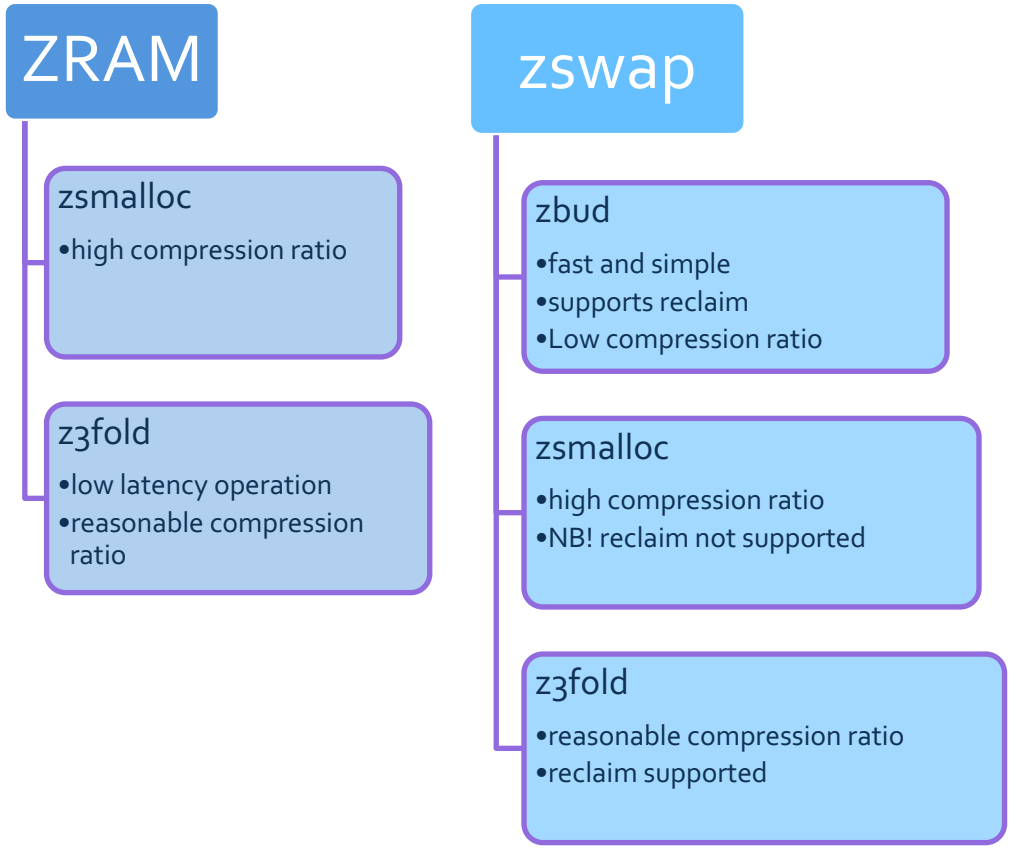
Why z3fold?

Konsulko Group

- For zswap
  - Supports reclaim unlike zsmalloc
  - Provides better compression than zbud
  - Scales well to multicore systems
- For ZRAM
  - Low latency operation
  - Reasonable compression ratio
  - Well-behaving on HMP (big.LITTLE®) systems

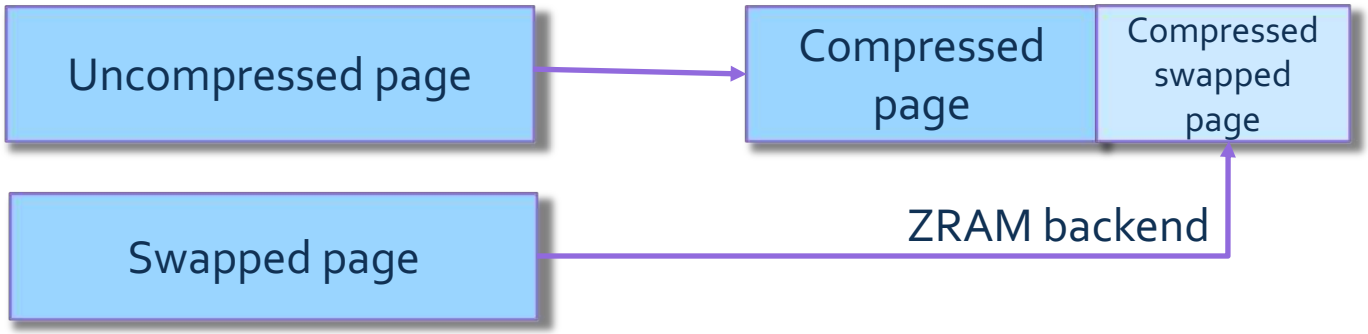
zsmalloc /zbud/z3fold: applicability

Konsulko Group



A new backend?

Konsulko Group



- Imagine hardware compression module
  - (De)compresses pages on-the-fly
  - Page "remainder" can be used for i.e. ZRAM
    - That would require a new backend
    - and backend-independent ZRAM

Conclusions so far

Konsulko Group

- 2 compression frontends, 3 (may get more) backends
  - That calls for unification
- Different compression backends have different things in focus
  - Depending on your goals, you may need any of these either for ZRAM or for zswap
- It's beneficial for the kernel ecosystem to have simple means to switch between backends for both ZRAM and zswap
- How does that match the current situation?

Backend compatibility matrix

Konsulko Group

	zbud	z3fold	zsmalloc
zswap			
zram	X	X	

ZRAM over zpool

zpool API

Konsulko Group

- zpool provides an abstract API for compressed allocators (*backends*)
- Mostly pass-through functions invoking callbacks of zpool clients
- zswap has already been converted to use zpool
- zbud, zsmalloc and z3fold implement zpool API
- ZRAM does not use zpool (yet? 😉)
  - Uses zsmalloc API directly instead

ZRAM and zpool?

Konsulko Group

- Why ZRAM doesn't use zpool (my guesswork)
  - There was no need
  - Another level of indirection
  - zpool API doesn't exactly match zsmalloc API
- Why these are not important enough
  - Now there is a need
  - The indirection is almost completely optimized out
  - zpool API can be extended
  - There may be more backends to come

ZRAM over zpool

Konsulko Group

- zpool API maps very well to zsmalloc API
- Only a few missing functions
  - `zpool_compact()` ← `zs_compact()`
  - `zpool_get_num_compacted()` ⇠ `zs_pool_stats()`
  - `zpool_huge_class_size()` ← `zs_huge_class_size()`
- Everything else has been live tested by using zswap with zsmalloc as a backend

ZRAM conversion example

Konsulko Group

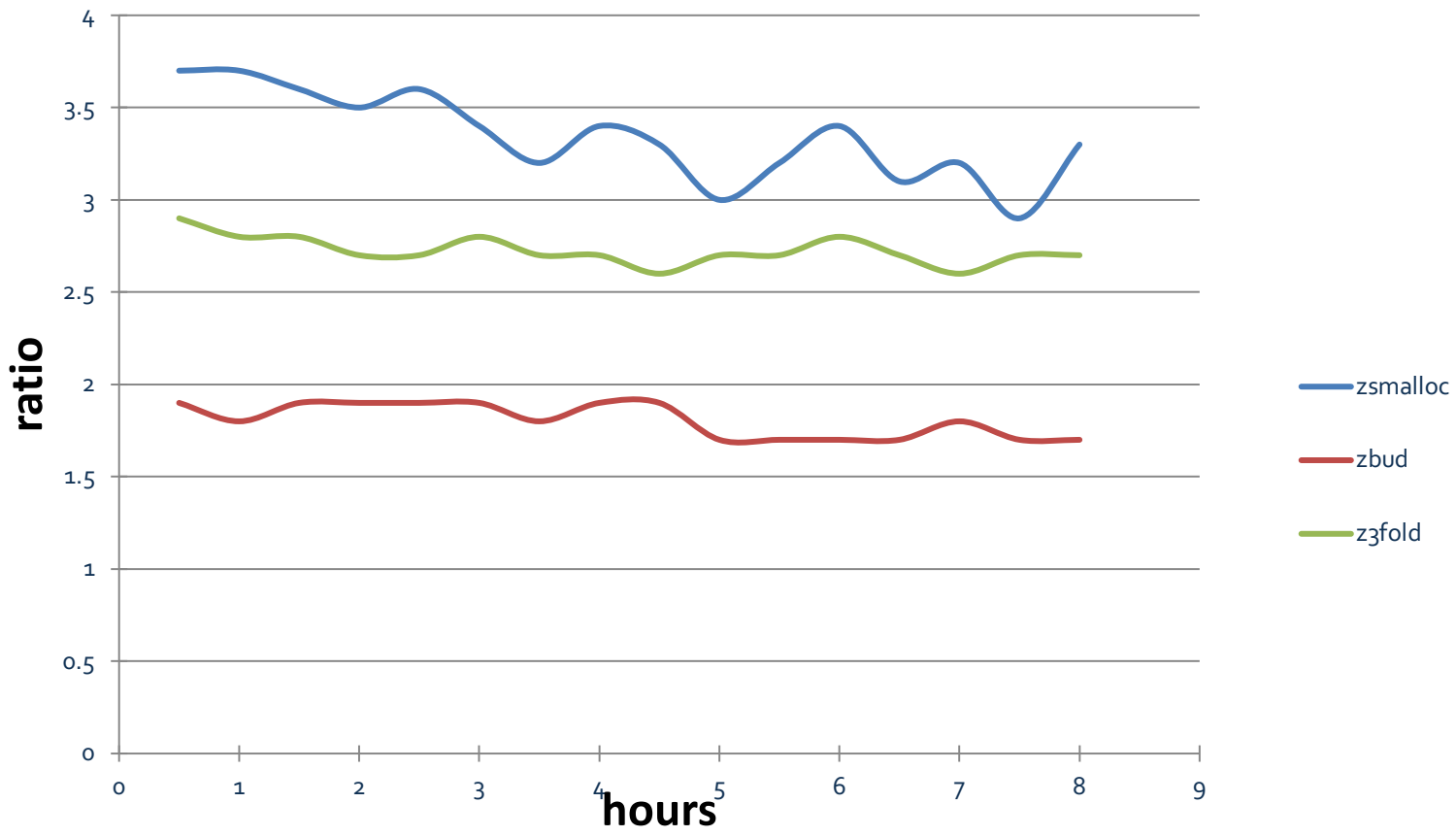
```
-       zs_free(zram->mem_pool, handle);
+       zpool_free(zram->mem_pool, handle);

        atomic64_sub(zram_get_obj_size(zram, index),
                        &zram->stats.compr_data_size);
@@ -1246,7 +1248,7 @@ static int __zram_bvec_read(struct zram *zram, struct page *pa,

        size = zram_get_obj_size(zram, index);

-       src = zs_map_object(zram->mem_pool, handle, ZS_MM_RO);
+       src = zpool_map_handle(zram->mem_pool, handle, ZPOOL_MM_RO);
        if (size == PAGE_SIZE) {
                dst = kmap_atomic(page);
                memcpy(dst, src, PAGE_SIZE);
@@ -1260,7 +1262,7 @@ static int __zram_bvec_read(struct zram *zram, struct page *pa,
                kunmap_atomic(dst);
                zcomp_stream_put(zram->comp);
        }
-       zs_unmap_object(zram->mem_pool, handle);
+       zpool_unmap_handle(zram->mem_pool, handle);
        zram_slot_unlock(zram, index);

        /* Should NEVER happen. Return bio error if it does. */
@@ -1335,7 +1337,7 @@ static int __zram_bvec_write(struct zram *zram, struct bio_vecbv
        if (unlikely(ret)) {
                zcomp_stream_put(zram->comp);
                pr_err("Compression failed! err=%d\n", ret);
-               zs_free(zram->mem_pool, handle);
+               zpool_free(zram->mem_pool, handle);
                return ret;
        }
```

Pretty straightforward huh?

Comparisons and charts

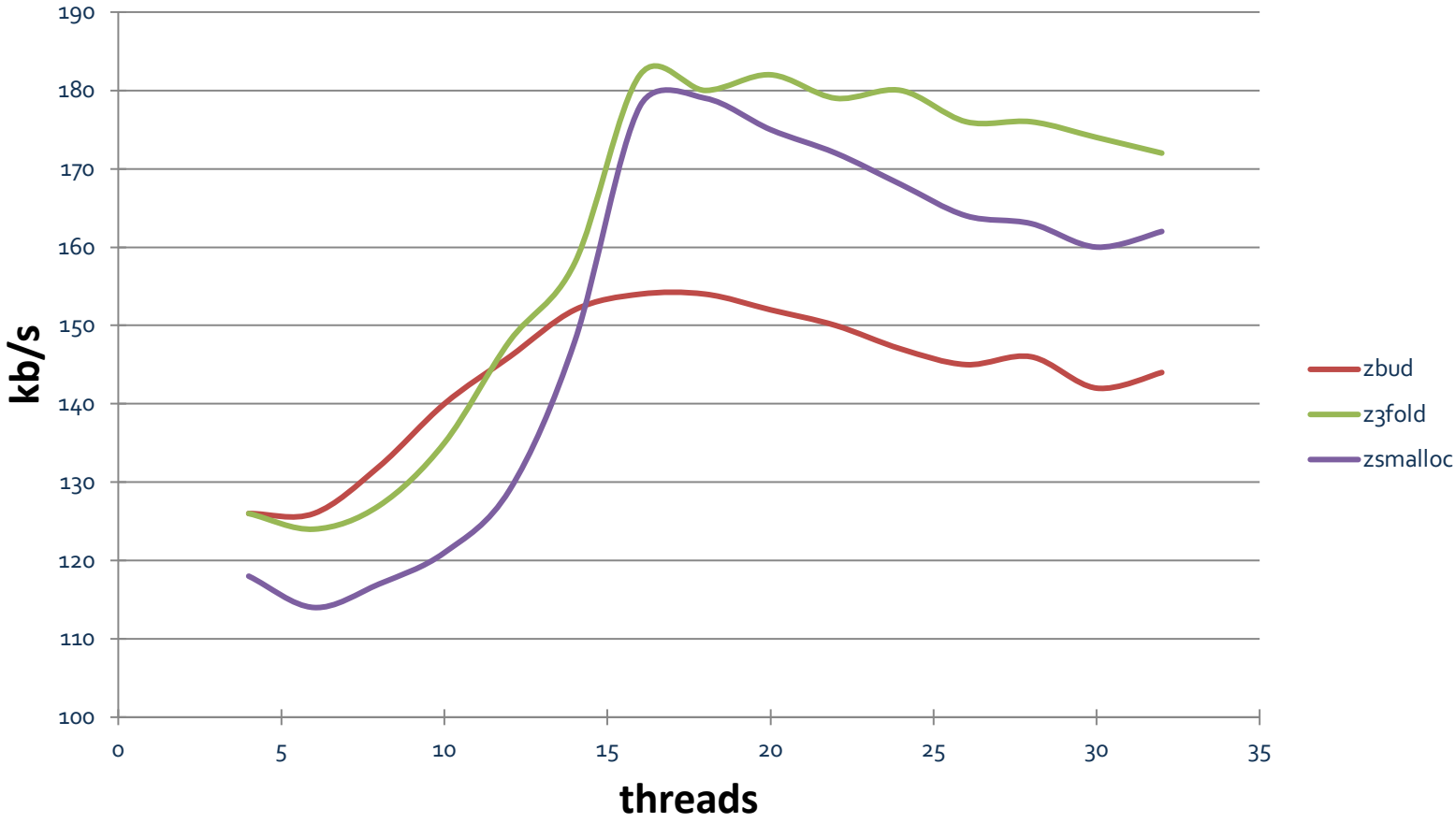
Compression under stress (as of kernel 4.9)

Konsulko Group



Random read-write performance comparison, kernel 4.19

Konsulko Group



Conclusions

Konsulko Group

- It's beneficial for the kernel ecosystem to have simple means to switch between backends for both ZRAM and zswap
  - So, ZRAM should be *decoupled* from zsmalloc
- zpool API is a good match for ZRAM to use as backend abstraction layer
  - Though not perfect, but we can work it out
- zpool API should be extended to fully match zsmalloc API
- ZRAM should be converted to use the extended zpool API

Questions?

Vitaly.Wool@konsulko.com