Allocators for Compressed Pages

Vitaly Wool
It’s an allocator, Cap.
- allocates memory according to user’s demands

It’s designed to store compressed data
- chunks of arbitrary length
  - usually quite small, way less than a page
  - ordinary kernel allocator would be a waste of space
- it doesn’t compress anything itself
Okay what purpose does all that serve?
Swapping

- using secondary storage to store and retrieve data
  - secondary storage is usually a HD or a flash device
  - saves memory by pushing rarely used pages out
- trade memory for performance?
  - reading and writing pages may be quite slow
Swapping optimization

• use RAM to cache swapped-out pages
  – but what’s the gain then?
• compress swapped-out pages
• trade performance for memory?
  – bigger cache means better performance
  – now we can be more flexible
Swapping and compression

• 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
Allocator for zswap?

- zbud: the first compressed data allocator
  - stores up to 2 objects per page
    - one bound to the beginning
    - one bound to the end
  - actual compression ratio may be quite low
    - imagine high amount of chunks sized 2K+Ɛ
zsmalloc

• came as an alternative to zbud
  – addresses the situation with 2k+ε sized objects
  – allocates objects contiguously within physically uncontiguous pages
  • objects may span across several pages
    – high compression ratio in the beginning
    – hard to mitigate in-page fragmentation over time as objects are allocated and released
Compressed allocator API

• 2 allocators used by zswap and doing the same thing differently
  – That calls for unification
• zpool: a common compressed allocator API
  – zswap is converted to use zpool
  – zbud and zswap both implement zpool API
Quite boring so far...
What happened next?
ZRAM: compressed RAM disk

• RAM block device with on-the-fly compression/decompression
  – uses zsmalloc directly via its API
• Alternative to zswap for embedded devices
  – no backing storage necessary
  – pages swapped to compressed RAM storage
Can’t do zram with zbud?!

<table>
<thead>
<tr>
<th></th>
<th>zbud</th>
<th>zsmalloco</th>
</tr>
</thead>
<tbody>
<tr>
<td>zswap</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>zram</td>
<td>❌</td>
<td>✓</td>
</tr>
</tbody>
</table>
ZRAM over zpool API

- **Pros**
  - unification and versatility

- **Cons**
  - none

- **Patches ready**

- **Several attempts to mainline the patches**
  - blocked by the maintainer
ZRAM over zpool API: test with zbud

• No performance degrade over time
  – stable and sustainable operation
• Peak performance lower than with zsmalloc
  – spinlocks don’t scale well
• Low compression ratio
  – 1.5x - 1.7x in real life scenarios
  – not enough to justify ZRAM for embedded
So what if we modify zbud to hold up to 3 objects?
z3fold: new kid on the block

- spun off zbud
- 3 objects per page instead of 2
- can handle PAGE_SIZE allocations
- only implements zpool API
  - no custom API here
- work started after ELC 2016 in San Diego
  - in the mainline since 4.8
z3fold: good for both ZRAM and zswap

• for ZRAM
  – supports up to page size allocations
  – low latency operation
  – good compression ratio

• for zswap
  – supports eviction unlike zmalloc
  – higher compression ratio than zbud
Ok let’s do the fun part. Comparisons!
Currently allowed combinations

<table>
<thead>
<tr>
<th></th>
<th>zbud</th>
<th>zsmallo</th>
<th>z3fold</th>
</tr>
</thead>
<tbody>
<tr>
<td>zswap</td>
<td>🟠</td>
<td>🟠</td>
<td>🟠</td>
</tr>
<tr>
<td>zram</td>
<td>🟥</td>
<td>🟠</td>
<td>🟥</td>
</tr>
</tbody>
</table>
Compression under stress (4.8)

- zsmallloc
- zbud
- z3fold

hours

ratio
Random read/write(4.8)

```
<table>
<thead>
<tr>
<th>threads</th>
<th>k</th>
<th>b/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>30</td>
</tr>
<tr>
<td>20</td>
<td>20</td>
<td>60</td>
</tr>
<tr>
<td>30</td>
<td>30</td>
<td>90</td>
</tr>
<tr>
<td>40</td>
<td>40</td>
<td>120</td>
</tr>
</tbody>
</table>
```

Graph showing kb/s vs threads for three different scenarios:
- zsmallloc
- zbud
- z3fold

The graph indicates a peak performance at around 30 threads for zsmallloc, followed by a decline, while zbud and z3fold show a more gradual increase and a slight decline at 40 threads.
Conclusions so far

- z3fold provides good compression ratio
- z3fold doesn’t scale well to larger number of CPUs/threads
  - Third level
    - Fourth level
      - Fifth level
z3fold: profiling

• using perf while running fio
  – identify bottlenecks under stress load
• using perf while Android LMK is triggered
  – how z3fold operation affects user experience
z3fold: profiling results

• spinlocks are the main obstacle to scalability
  – the “big” spinlock that protects “unbudded” lists is the biggest one
• using perf while Android LMK is triggered
  – how z3fold operation affects user experience
z3fold: per-page locks

- Keep “big” spinlock for list operations
- Have “small” spinlocks to protect in-page operation
  - this goes well with async in-page layout optimization
- in mainline kernel since 4.11
Random read/write(4.12)

- zsmalloc
- zbud
- z3fold
- z3fold 4.12

**kb/s**

**threads**
z3fold: lockless lists (llist)

• Idea: implement unbuddied lists using llist
  – Should improve scalability with less locking needed

• Unfortunately llist wasn't a fit
  – Can't do a llist_del
    • Complicates unbuddied lists manipulation up to the point where it makes no sense
z3fold: per-CPU “unbuddied” lists

• z3fold can operate only on this CPU's list
  – Reduces contention on spin lock
  – Speeds up search

• That can have adverse effect on ratio
  – Z3fold header gets bigger
  – Worse selection
  – More memory for multiple lists

• Will get into 4.14
Random read/write (4.14-rc4)

![Graph showing random read/write performance with threads on the x-axis and kb/s on the y-axis, with lines for z3fold 4.14, z3fold, and z3fold 4.12.]
z3fold: bit locks

- Z3fold header size better be 1 chunk
  - Now 2

- Bit locks may be used to mitigate bigger header
  - Slightly worse performance
  - Evaluation in progress
Conclusions

- Z3fold is still a young allocator
- Still z3fold already outperforms other allocators
- Z3fold is a good fit both for zswap and ZRAM
- We need to push ZRAM to use zpool
Questions welcome!

vitalywool@gmail.com