Decoupling ZRAM from a specific backend

Vitaly Wool, Konsulko Group
About me

- 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

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

- 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

- 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 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

- 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)

- 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)

```
We take it
```

```
\begin{itemize}
  \item $L_N$
  \item $N$ free chunks
  \item $N + 1$ free chunks
  \item $N + 2$ free chunks
\end{itemize}
```

```
list_head
```

Unbudded lists
zsmalloc (allocator for ZRAM)

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

Diagram:

- Pages
- zspage
- obj
- obj
- obj
- empty
zmalloc and zbud: applicability

**ZRAM**
- **zmalloc**
  - high compression ratio
  - reclaim not used

**zswap**
- **zbud**
  - fast and simple
  - supports reclaim
- **zmalloc**
  - high compression ratio
z3fold: new kid on the block

- 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?

- For zswap
  - Supports reclaim unlike zmalloc
  - 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
zsmallloc /zbud/z3fold: applicability

ZRAM

zsmallloc
- high compression ratio

z3fold
- low latency operation
- reasonable compression ratio

zswap

zbud
- fast and simple
- supports reclaim
- Low compression ratio

zsmallloc
- high compression ratio
- NB! reclaim not supported

z3fold
- reasonable compression ratio
- reclaim supported
A new backend?

- 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

- 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

<table>
<thead>
<tr>
<th></th>
<th>zbud</th>
<th>z3fold</th>
<th>zsmalloc</th>
</tr>
</thead>
<tbody>
<tr>
<td>zswap</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>zram</td>
<td>✓</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>
ZRAM over zpool
zpool API

- 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?

- 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

- 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

Pretty straightforward huh?
Comparisons and charts
Compression under stress (as of kernel 4.9)
Random read-write performance comparison, kernel 4.19

![Graph showing performance comparison for different thread counts and kernel versions. The graph plots kb/s against threads, with three lines representing 'zbud', 'z3fold', and 'zsmalloc'.]
Conclusions

- 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