Selective Rejection Technique for Compressed Object at Compressed Swap Cache

Md Salahuddin and Dongkun Shin

Sungkyunkwan University, Suwon, Korea

sagor@skku.edu, dongkun@skku.edu

***Abstract* -** The compressed swap cache such as zswap is a memory overcommitment technique of Linux kernel, which is suitable for smart devices with NAND flash swap storage. It compresses the pages which are in the process of being swapped out and stores them in a RAM-based memory pool. Since the sizes of compressed pages vary widely, the memory pool should efficiently manage the memory area in order to minimize memory wastage and flash memory write operations. We propose an enhanced management technique for the zswap memory pool. Experiment results show that the improved memory pool reduces the response time of memory-hungry applications by up to 11.3% and takes up to 46% less swap space than the default memory pool.

Keywords: Memory, Zsmalloc, Swap Cache, Linux kernel, Embedded Systems.

1 Introduction

Modern smart devices like smartphones and IoT devices require a large capacity of main memory to support complex applications and to utilize the multi-core hardware. Device manufactures are increasing the heap size limit of each application as well as the total DRAM size in order to support more versatile applications. However, it is not easy to increase the memory size since these devices are sensitive to power consumption, device size and cost. For example, Samsung IoT module ARTIK 5 is equipped with 512MB of DRAM. The Tizen smartphone Samsung Z1 has only 768 MB of DRAM. Therefore, researchers are exploring other solutions to run memory-hungry applications on such low-cost devices.

Smart devices generally have NAND flash-based storage such as embedded MultiMediaCard (eMMC) as internal storage and Secure Digital (SD) card as external storage. These storage devices are unsuitable for swap space due to the long access latency and limited endurance of flash memory. Therefore, when there is no sufficient free memory for running applications, Linux's Out-Of-Memory (OOM) killer terminates several processes to reclaim the memory space allocated for the processes. The compressed swap cache technique can be useful to increase the available memory size without killing background processes on embedded devices [1][2]. For a swap-out request, the compressed swap cache technique compresses the page and writes it at the reserved memory space. For a swap-in request, it decompresses the page and loads it at the page cache. Since the reads from the compressed cache are faster than the reads from a swap storage device, the performance impact of swapping can be mitigated. Moreover, it can reduce the life-shortening writes on the NAND flash storage as swap device. Since the size of the compressed page varies widely, the memory pool for the compressed objects should be managed efficiently. A compressed swap cache can be efficient if it generates less swap write operations and increases the swap cache utilization. However, the current memory pools do not satisfy both requirements.

In this paper, we propose several enhancements of the *zsmalloc* memory pool of Linux. The techniques can increase the density of compressed swap cache. From experiments, we demonstrate that the enhanced zsmalloc can reduce the response times of memory-hungry applications by up to 11.3% and takes up to 46% less swap space compared to the Linux's default compressed memory pool, *zbud*.

2 Linux's compressed memory system

Currently, three in-kernel memory compression solutions exist in Linux kernel [3]: *zram*, *zcache* and *zswap*. The zram is a RAM based block device for swap pages. When the swap subsystem sends a page to the zram device, the page is sent to the zram driver through the block I/O subsystem. Then, the page is compressed and is saved at the RAM-based block device. Since the swapped-out page is written at memory, zram provides fast response time. In addition, since the page is compressed, zram can save memory space.

On the other hand, zcache and zswap use the frontswap of swap subsystem. The swapped pages go directly to zcache/zswap and thus the swap-out has a low latency [3]. Each swap page is compressed into a compressed page, called *zpage*. Each zpage is associated with a key which is generated with the page address. The key is searched at swap-in operation. User can control the maximum size of the compressed pool. Since zcache can store both swap data and page cache data, it requires a large size of index key. However, zswap handles only swap data and thus it is more suitable for resource-constrained embedded systems.

Zswap uses the dynamically allocated memory pool, called *zpool*, in order to store zpages. There are two implementations of zpool in the current Linux kernel: *zbud* and *zsmalloc*. Zbud is the default zpool for zswap. It stores compressed pages in pairs in a single memory page called *zbud page*. The zbud type zpool allocates exactly one page to store two compressed pages, which means the compression ratio will always be 0.5 or worse because of half-full zbud pages. Zbud maintains an LRU list of zbud pages to select victims for writeback operations when the memory pool is full. Zbud provides a worse density than zsmalloc due to many internal fragmentations in the allocated zbud pages. Therefore, the memory pool reaches its size limit quickly.

Zsmalloc works well under low memory conditions since it maintains multiple classes to store different sizes of compressed pages. Zsmalloc links several 4KB of pages and forms a compound page, called *zspage*. The compressed pages can be stored within the zspage across the 4KB-page boundaries. In order to manage zpages efficiently, zsmalloc divides the memory pool into several classes, called *size class*, each of which can store a different size of compressed pages, as shown in the Figure 1. For example, the class 2 has 12KB size zspages and stores the compressed pages less than 48 bytes. One zspage can store up to 256 compressed pages. Currently, zsmalloc supports 69 number of size classes. Each size class maintains zspages in different *fullness groups* depending on the number of live objects they contain. When allocating or freeing objects, each zspage is assigned to the appropriate fullness group. For example, the fullness status of the zspage can change from almost full group to almost empty group when freeing an object.

Zpool can shrink the actual memory size of the pool by evicting some zpages. The fullness group is maintained to assist the pool shrinking feature of zsmalloc, which is not implemented yet. For this reason, once zswap fills it cannot evict the oldest page, it can only reject new pages. Zsmalloc shows poor performance in such a situation. Therefore, zbud is now preferable for zswap when the memory pool is small although zsmalloc provides higher density than zbud. Yang *et al*. [4] proposed a software-based RAM compression technique called CRAMES. The memory manager of CREAMS is built upon the kernel page allocator. In order to identify the most appropriate memory allocation, they evaluated several allocation algorithms for the kernel page allocator. However, CREAMS suffers from internal fragmentation and shows poor performance in low memory situations. Therefore, an enhanced memory pool is required to provide high performance and high density of compressed cache.

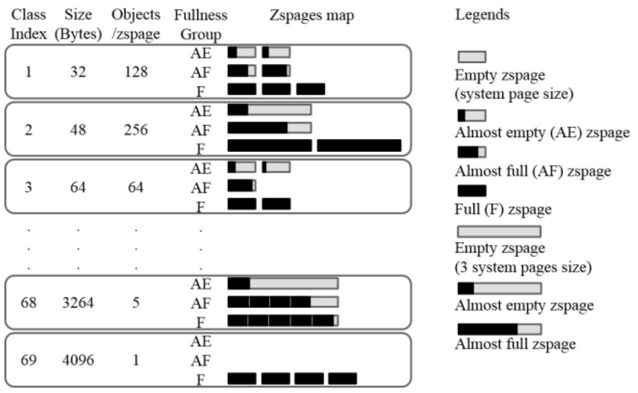

Figure 1. The memory pool of zsmalloc.

3 Enhanced zsmalloc

We propose some improvements for zsmalloc in order to satisfy high performance and high density requirements. Zsmalloc should writeback some old pages into the swap storage to create free memory space for a new zpage store request. Then, it can store more recent zpages which have high probabilities to be referenced again. In order to free a zspage, zsmalloc should evict all objects in the zspage. Our technique maintains the least recently used (LRU) list of all zspages of each fullness group of each class. However, the LRU ordering between size classes is not maintained since our observation shows that the access pattern for size classes follows the uniform distribution. Therefore, we select the victim class in a round-robin manner. Table 1 shows the pseudocode of our zsmalloc writeback algorithm.

Table 1. Zsmalloc writeback pseudocode.

```
Initialize static Marker to zero
Input: pool, count
Output: reclaimed
1.    while reclaimed is less than count
2.        do if FullnessGroup[Marker] is not empty
3.            then get head zspage from FullnessGroup[Marker]
          of given pool // Head of the group is the LRU zspage
4.            reclaimed ← evict the selected zspage
5.        Marker ← (Marker + 1) % n
          // n is the total number of fullness group of that pool
6.    return reclaimed
```

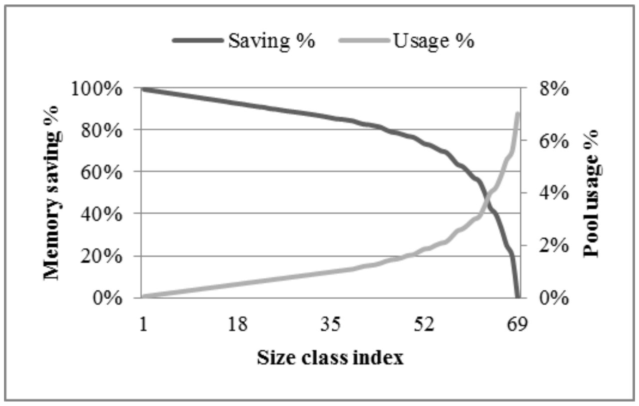

Figure 2. Analysis of size class utilization in zsmalloc.

Current zsmalloc implementation accepts the zpage size up to system page size (4K bytes in 32 bit Linux system). Figure 2 shows the distributions of memory savings and memory pool usages of different size classes. The high indexed size classes show poor memory savings and high memory usages. For example, if zsmalloc stores one zpage in the 68th size class, it can save 20% of system page size memory while the first two size classes give more than 99% of memory savings. Therefore, it is better to reject zpages in high indexed size classes to increase the zpool utilization when the memory pool has a low free space. In this way, more number of small size of zpages can be stored. As a result, the memory pool density will be increased and the hit ratio of compressed cache will also be increased.

In order to decrease the maximum size of allowed zpage, our technique rejects all the zpages greater than the allowed size. If the allowed size is too small, the memory saving is significant but more zpages should be written at the swap storage. Therefore, a proper threshold value should be selected. As shown in Figure 3, when the maximum size of allowed object is 3072 bytes, the swap space usage is minimized. If the size becomes less than 3072 bytes, the swap space usage increases since zsmalloc should reject all large objects. For this reason, we set the maximum size of allowed zpage to 3072 bytes.

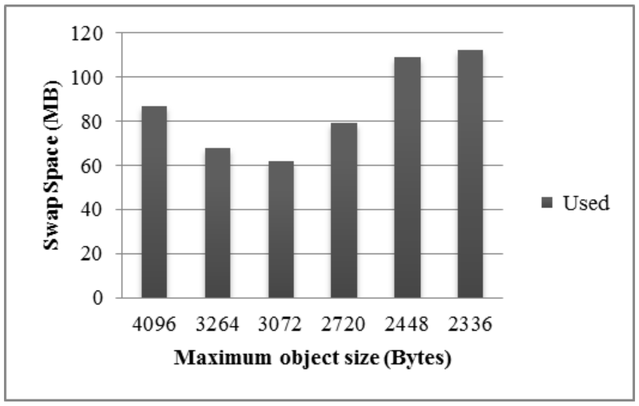

Figure 3. Swap usage with different maximum object size.

4 Experiments

For experiment, we used an Odroid-U3 development board equipped with Exynos 4412 CPU and 2GB of DRAM. It runs Tizen 2.2 over Linux kernel 3.10.52. We have back-ported zpool, zbud, zsmalloc and zswap from the mainline Linux kernel 4.0.5. Then, we implemented our writeback operation of zsmalloc and set the maximum allowed zpage size to 3072 bytes. The maximum compressed pool size is set to 20MB, and 256MB of swap space is reserved at SD card. We measured the execution time of a memory hungry application while running the Tizen web browser which opens several web pages. The memory hungry applicaton allocates 1500 MB of heap memory. In order to allocate such a big memory, many swap-out pages are sent to zswap. The swap space usage, and the density and write-back page count of the zswap memory pool are measured.

Table 2 compares the memory allocation time, the swap space usage, the memory pool density and the write-back page count for allocating 1500 MB of memory at different schemes, the original zsmalloc, zbud and the enhaned zsmalloc. The enhanced zsmalloc shows about 11.3% shorter execution time than zbud, it also shows about 73.5% shorter execution time than the current zsmalloc. It also takes about 46% and 51% less swap space than zbud and the original zsmalloc, respectively.

Since zbud consumes the memory pool quickly, the kernel page allocator is frequently called by zbud. However, if there is no available free memory in the system, the page allocator fails to allocate memory for zbud, and therefore zbud is forced to reject an incoming compressed page, which is eventually written at the swap storage device. Therefore, the enhanced zsmalloc can show better performance and use less swap space than zbud.

The enhanced zsmalloc pool density shows a higher density than other schemes. As the scheme rejects large size of zpages, it can store more number of small sized zpages in the zsmalloc memory pool. Since the pool density of zbud is low, zbud shoud writeback many old zpages to the swap storage space for new zpage requests. Therefore, the write-back page count of zbud is much higher than the enhanced zsmalloc. As the original zsmalloc does not have the write-back functionality, its write-back page count is zero.

Table 2. Performance comparison of different zpool.

Factors	Original zsmalloc	Zbud	Enhanced zsmalloc
Time (s)	28.04	17.99	16.16
Swap usage (MB)	127	115	62
Density (Obj/MB)	619	499	641
Write-back count	0	19128	1911

5 Conclusions

An efficient utilization of memory is crucial for embedded systems. In this paper, we discussed the compressed cache techniques of Linux kernel. We proposed an enhanced zsmalloc which rejects large size of compressed pages to provide better performance and use less swap space than zbud. The enhanced zsmalloc shows better performance while it takes less swap space than zbud and the current zsmalloc. The enhanced zsmalloc also increases the pool density compared to zbud.

Acknowledgements

This work was supported by Samsung R&D Institute Bangladesh and DMC R&D Center, Samsung Electronics Co. Ltd.

References

[1] A. F. Briglia, A. Bezerra, L. Moiseichuk, and N. Gupta, "Evaluating effects of cache memory compression on embedded systems", *Proceedings of the Linux Symposium*, Vol. 1, pp. 53-64, Jun. 2007.
[2] S. Jennings, "The zswap compressed swap cache", https://lwn.net/Articles/537422/, 2013.
[3] D. Maegenheimer, "In-kernel memory compression", https://lwn.net/Articles/545244/, 2013.
[4] L. Yang, R.P. Dick, H. Lekatsas, and S. Chakradhar, "Online memory compression for embedded systems", *ACM Transactions on Embedded Computing Systems*, Vol. 9, No. 3, pp. 27:1 – 27:30, Feb. 2010.