CHINA LINUX KERNEL
中国Linux内核开发者大会

vivo

ZRAM 多压缩算法效率实践与评估

汪劼文
vivo性能优化

CHINA LINUX KERNEL
中国Linux内核开发者大会

vivo

目录

CONTENTS

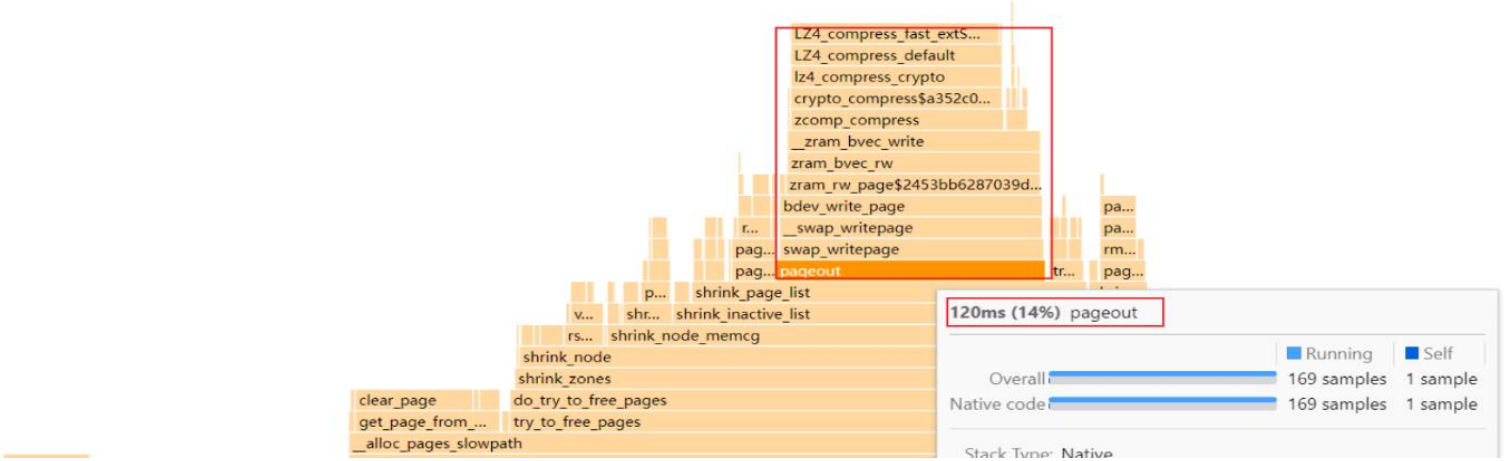
Part1：背景

Part1：背景

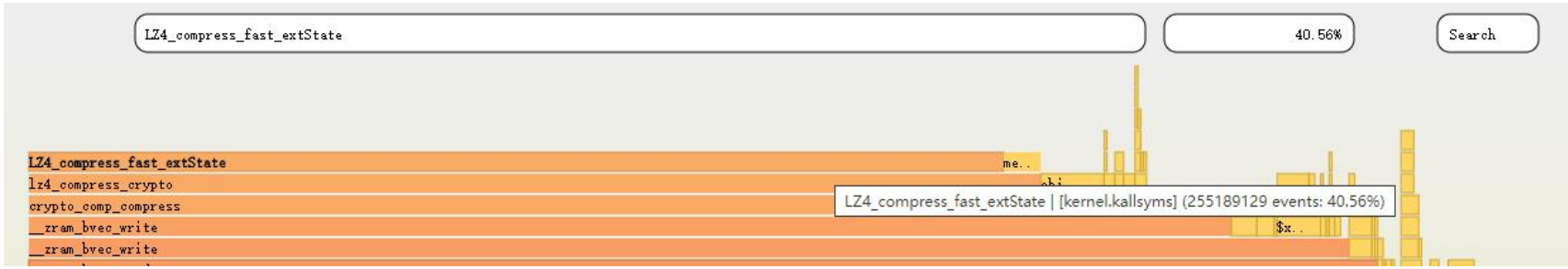
系统多后台场景（高负载），应用启动时延明显且波动大

Part1：问题拆解

- 相机启动时，direct reclaim/slowpath 在启动流程占比约 14% ，如果加上 kswapd 负载，会更高。



- 内存回收中压缩在内存回收占比，约 40%+

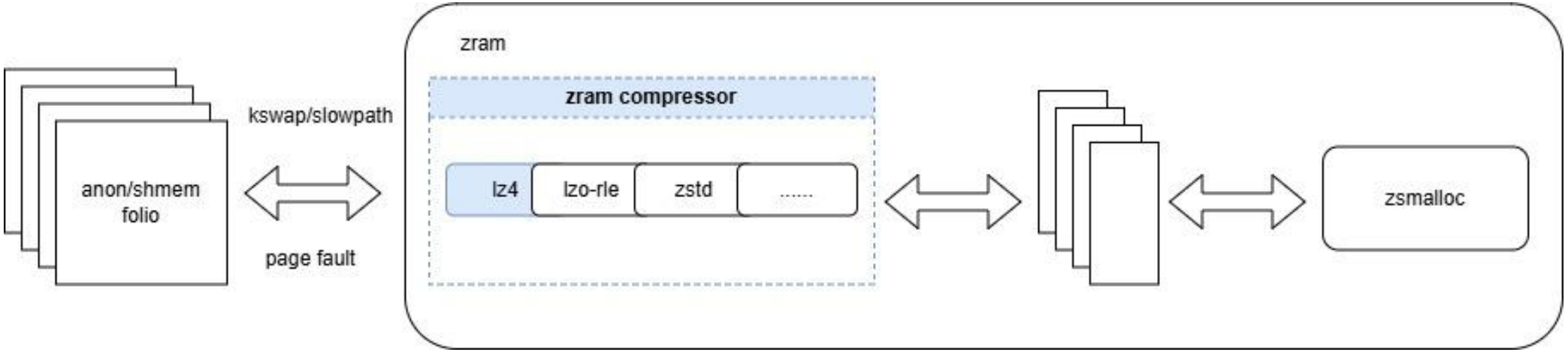


方向：优化压缩。

Part1：zram 简介

zram: Compressed RAM-based block devices

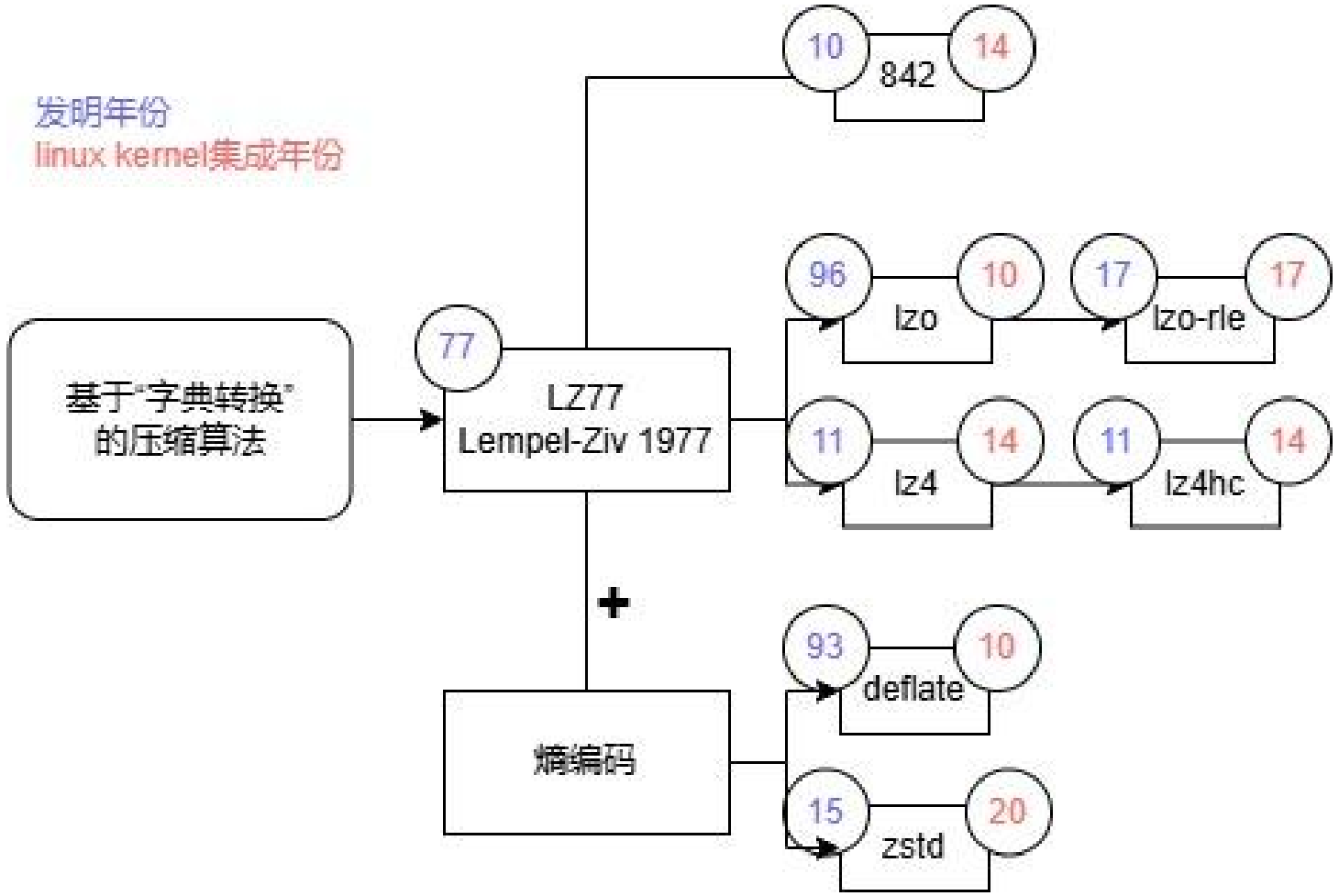
- 块设备，配置为 swap 分区
- 压缩/解压 PAGE_SIZE 数据，存入 zsmalloc
- 压缩算法：多种可选配置，初始化时配置一种固定的压缩算法



Part2：压缩算法及评估

Part2：zram 常用无损压缩算法

● zram 当前支持的开源压缩算法：lz4、lzo-rle、lzo、lz4hc、zstd、842、deflate。



Part2：LZ系列压缩算法原理简介

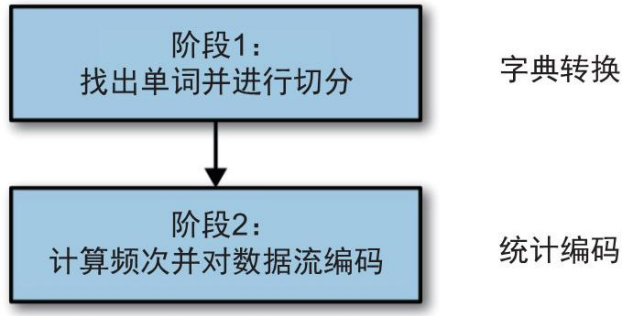
压缩：找出重复字符，用更短的字符替代

● 基于“字典转换”　　核心问题：如何找到合适的重复字符？理想分词问题

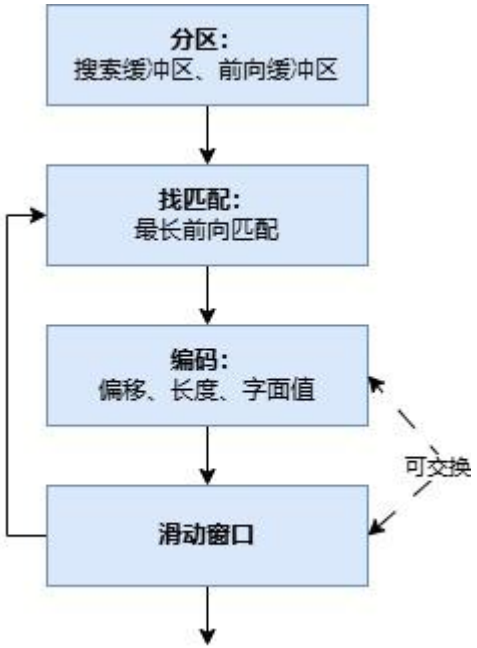
TOBEORNOTTOBEORTOBEORNOT

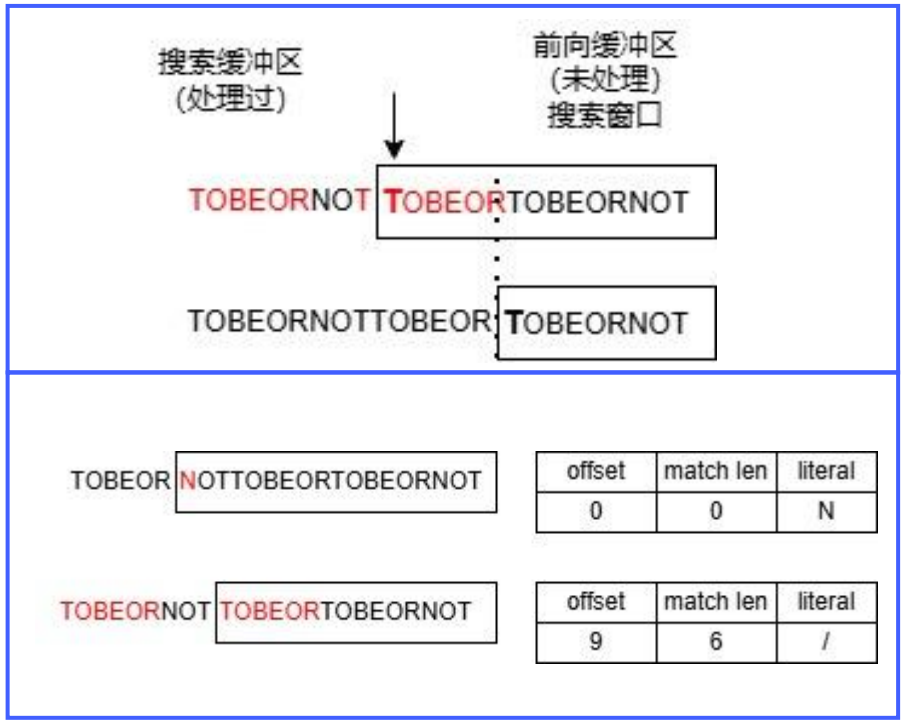
TOBEORNOTTOBEORTOBEORNOT：2bit*11=22bit

TOBEOR NOT TOBEOR TOBEOR NOT=1bit*5=5bit



● LZ　　1977 年两位研究员 Abraham Lempel 和 Jacob Ziv 提出了几种“理想分词”的方法，根据提出年份命名为 lz77、lz78。

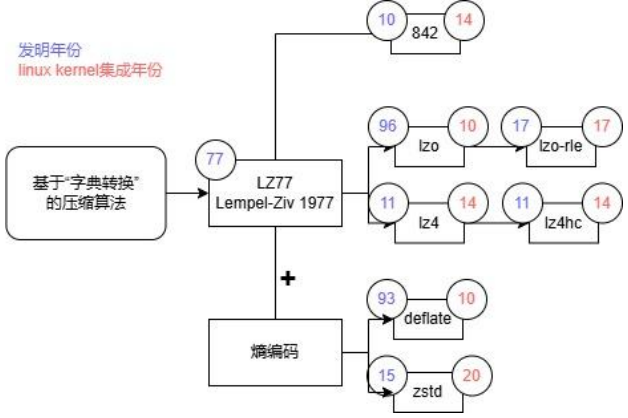




搜索缓冲区	前向缓冲区	输出
	TOBEORNOTTOBEORTOBE	0,0,T
T	OBEORNOTTOBEORTOBE	0,0,0
…	…	…
TOBE	ORNOTTOBEORTOBE	3,1
TOBEO	RNOTTOBEORTOBE	0,0,R
…	…	…
TOBEORNOT	TOBEORTOBE	9,6
TOBEORNOTTOBEOR	TOBE	15,4

参考：《数据压缩入门》

Part2：压缩算法理论评估



CHINA LINUX KERNEL
中国Linux内核开发者大会

● 各个无损压缩算法的原理差异核心。**空间**　　**时间**，不可兼得。

压缩率：匹配更准、编码更省。

速度：匹配简化，编码利于读取，滑窗更快，多进程。

	原理特点	压缩率	压缩速度	解压速度
lz4	匹配：优化 hash，冲突时直接覆盖 编码：顺序访问，字段按字节对齐	有牺牲	快	！！！！！
lz4hc	匹配：更深入搜素	！！！	很慢	同 lz4
lzo	匹配：hash 链表。 编码：变长编码、存在按位操作	比 lz4 好	略差于 lz4	差于 lz4
lzo-rle	针对连续重复字节优化	与 lzo 相当	略好于 lzo	略好于 lzo
Lz77+熵编码				
deflate	lz77+huffman编码 常用于 gzip（.gz 文件）中	！！！！	–	–
zstd	lz77+FSE有限状态熵/HUFF facebook 推出，天生多线程	！！！！！	deflate < zstd << lz4 多线程可加速	
厂商				
842	IBM			
wkdm	苹果在内存回收中使用			

优势

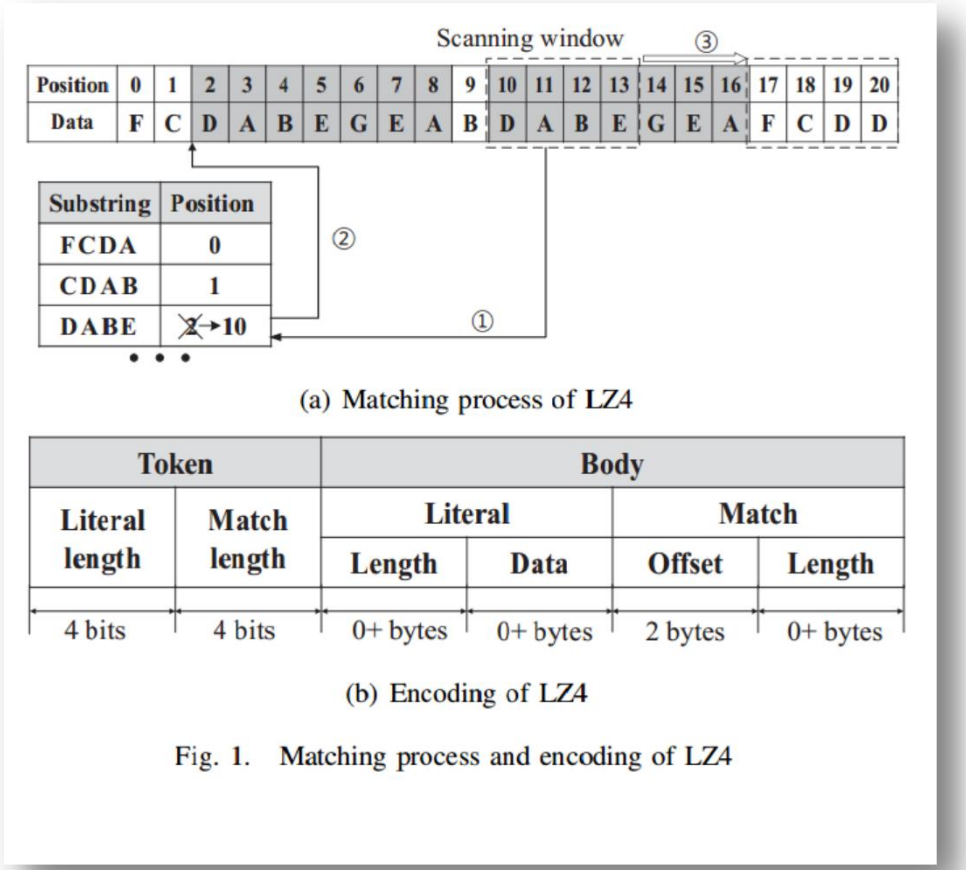
劣势

Part2: LZ4M

LZ4m: A fast compression algorithm for in-memory data。

- **分词**：匿名内存数据一般按 4byte 对齐。以 4byte 为单位，做 hash 填表、匹配、滑动窗口。 => 压缩速度↑，压缩率↓
- **编码**：减少存储 offset 大小。2bytes 可以存储数据长度 65536，内存压缩数据一般以 PAGE(4096) 单位，12bit 即可。=> 压缩率↑

预期结果：仅用于内存数据压缩情况。压缩速度更快，对压缩率影响小。

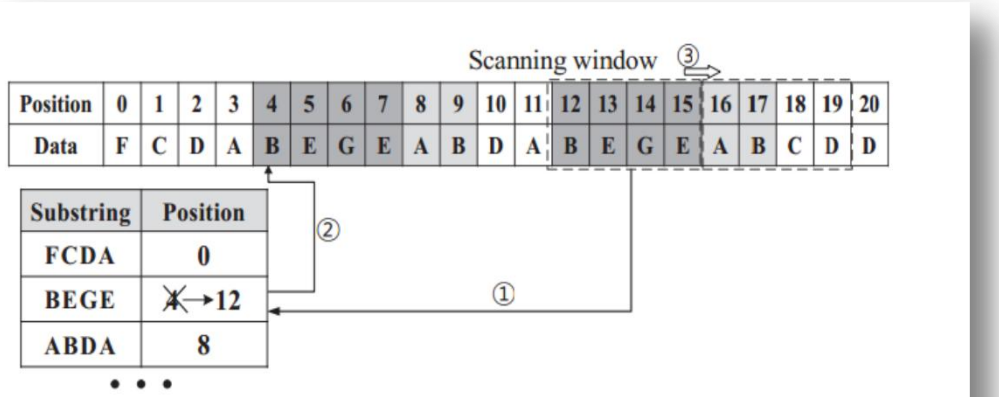


(a) Matching process of LZ4

Token		Body			
Literal length	Match length	Literal		Match	
		Length	Data	Offset	Length
4 bits	4 bits	0+ bytes	0+ bytes	2 bytes	0+ bytes

(b) Encoding of LZ4

Fig. 1. Matching process and encoding of LZ4



(a) Matching process of LZ4m

Token			Body			
Offset	Literal length	Match length	Literal		Match	
			Length	Data	Offset	Length
2 bits	3 bits	3 bits	0+ bytes	0+ bytes	1 byte	0+ bytes

(b) Encoding of LZ4m

Fig. 2. Matching process and encoding of LZ4m

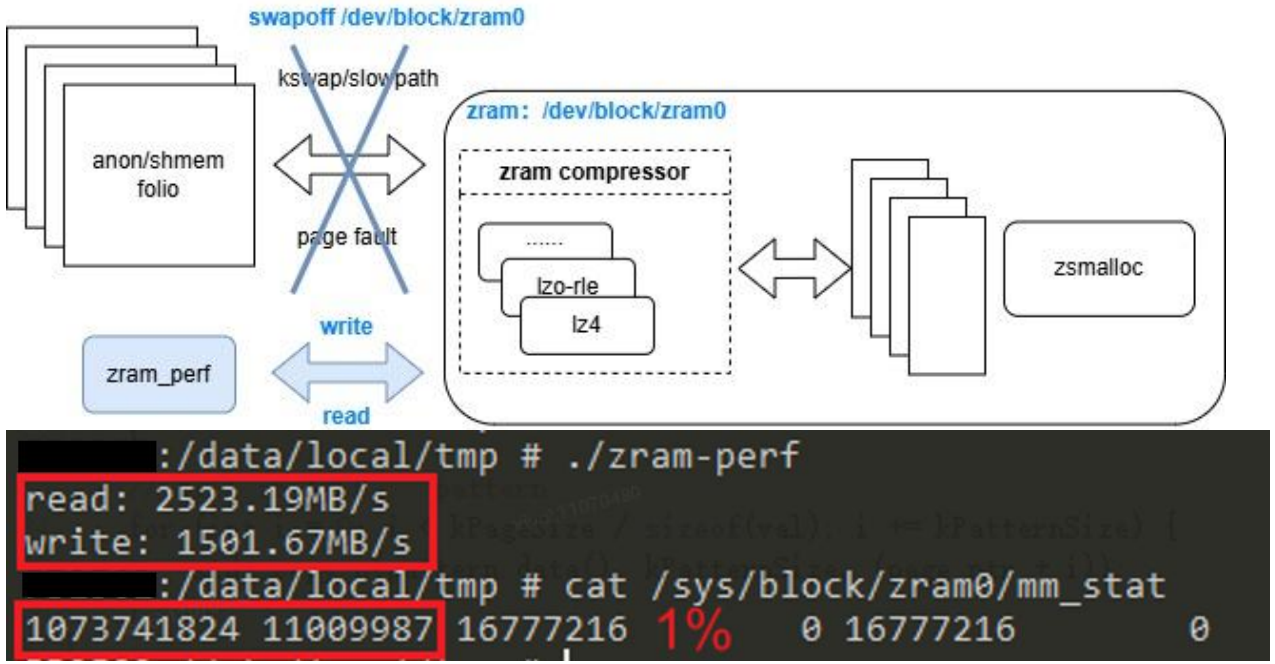
参考：https://ieeexplore.ieee.org/document/7889380/

Part2：Android zram 效率评估实践

指标：压缩速度、解压速度，压缩率。 **影响因素：**算法、数据源、CPU（绑核绑频）

目标：评估 Android 手机上 zram 效率

- 工具选择：~~lzbench(仅压缩算法)~~，zram-perf



- 数据源选择：（问题：为什么项目中达不到评估的速度？）
  - Silesia compression corpus 无损压缩语料库。
  - zram-perf: uint32_t val = rand() & 0xfff; pattern[i] = val + i;
  - zram-dump：多应用连续启动尽量塞满 swap，导出此刻 swap。

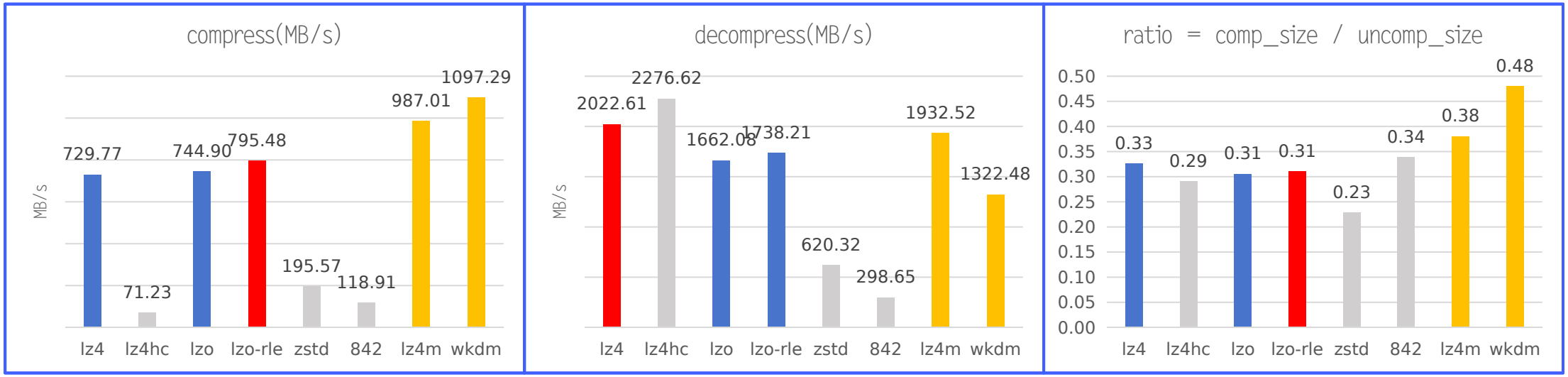
The benchmark uses lzbench, from @inikep compiled with GCC v8.2.0 on Linux 64-bits (Ubuntu 4.18.0-17). The reference system uses a Core i7-9700K CPU @ 4.9GHz (w/ turbo boost). Benchmark evaluates the compression of reference Silesia Corpus in single-thread mode.

Compressor	Factor	Compression	Decompression
memcpy	1.000	13700 MB/s	13700 MB/s
LZ4 default (v1.9.0)	**2.101**	**780 MB/s**	**4970 MB/s**
LZO 2.09	2.108	670 MB/s	860 MB/s
QuickLZ 1.5.0	2.238	575 MB/s	780 MB/s
Snappy 1.1.4	2.091	565 MB/s	1950 MB/s
Zstandard 1.4.0 -1	2.883	515 MB/s	1380 MB/s
LZF v3.6	2.073	415 MB/s	910 MB/s
zlib deflate 1.2.11 -1	2.730	100 MB/s	415 MB/s
LZ4 HC -9 (v1.9.0)	**2.721**	41 MB/s	**4900 MB/s**
zlib deflate 1.2.11 -6	3.099	36 MB/s	445 MB/s

参考：https://github.com/lz4/lz4

Part2：基础指标

- zram-perf 输入源改为 zram-dump 数据，绑定cpu大核最高频，单进程同步，测试各压缩算法效率。



- 当前内存回收常用：满足基础压缩解压速度。lz4（解压速度快），lzo-rle（压缩率更优）

  单项指标太差不可用：压缩速度：lz4hc、842、zstd，解压速度：zstd、842

- lz4m，同预期，速度快但压缩率有劣化

- wkdm，压缩率最差，压缩速度最快

Part3：内存指标-腾内存速度

问题：1. 压缩算法各有优劣，应该以什么标准选择？

压缩率：可用内存，后台保活。　压缩速度：slowpath/kswapd 耗时。　解压速度：page fault 不卡。

2. wkdm 的压缩速度快但压缩率不好为什么能落地产品？

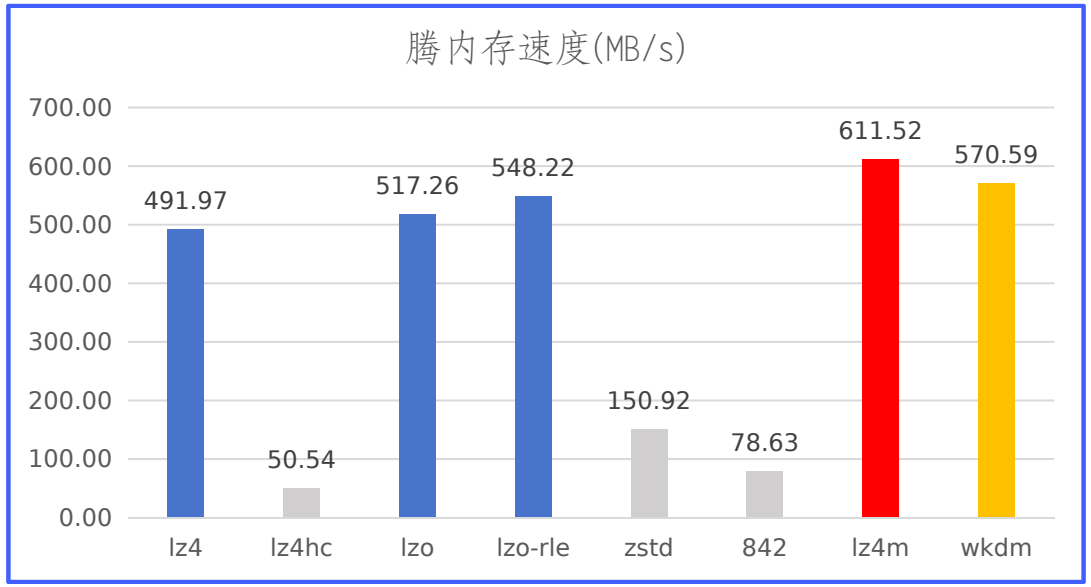
回顾问题背景，高负载时应用启动慢，进 slowpath 耗时长。目标：尽快腾出内存，结合压缩率和压缩速度。

- **腾内存速度 = comp_speed * (1 - ratio)**，单位时间可以腾出多少内存

- lz4m 腾内存速度最快，可用于**内存需求紧急的场景**：前台进程进 slowpath。

问题：压缩率差导致总体腾出的可用内存变少。

假设压 8G 数据，可用内存则少腾出约 8G*(0.38-0.33)=8G*5%=446MB，约2-3个保活。



Part3： ZRAM_MULIT_COMP

Part3：ZRAM_MULIT_COMP-配置

23 年初 kernel 6.2 引入 ZRAM_MULIT_COMP 特性，24 年 ard15 google 推荐使用。

参考：https://lore.kernel.org/all/20221109115047.2921851-5-senozhatsky@chromium.org/T/#m67546f76e537d8c5532d27496f8bcfd0f735a5da

配置：

- 压缩相关结构体从 1 个改为 4 个，可设置1个主压缩算法（PRIMARY），3个次压缩算法(SECONDARY)。当前已有的压缩、解压、读取、设置均修改为使用 主压缩算法。

comp_algs：压缩算法名字，zram 初始化前可通过写 comp_algorithm 节点修改。

comps：zram 压缩相关的结构体，在 zram 初始化 disksize 时根据以上指定的压缩算法初始化。

- echo "algo=zstd" > /sys/block/zramX/recomp_algorithm

新增节点配置重压缩的压缩算法，最多3种。可选配置：priority

修改该节点时仅记录算法名，disksize配置时做初始化

```
+#ifdef CONFIG_ZRAM_MULTI_COMP
+#define ZRAM_PRIMARY_COMP      0U
+#define ZRAM_SECONDARY_COMP    1U
+#define ZRAM_MAX_COMPS 4U
+#else
+#define ZRAM_PRIMARY_COMP      0U
+#define ZRAM_SECONDARY_COMP    0U
+#define ZRAM_MAX_COMPS 1U
+#endif
+
 struct zram {
        struct zram_table_entry *table;
        struct zs_pool *mem_pool;
-       struct zcomp *comp;
+       struct zcomp *comps[ZRAM_MAX_COMPS];
        struct gendisk *disk;
        /* Prevent concurrent execution of device init */
        struct rw_semaphore init_lock;
@@ -107,7 +117,7 @@ struct zram {
         * we can store in a disk.
         */
        u64 disksize;   /* bytes */
-       char compressor[CRYPTO_MAX_ALG_NAME];
+       const char *comp_algs[ZRAM_MAX_COMPS];
```

Part3：ZRAM_MULIT_COMP-管理

如何确认一个 zram slot 使用了哪种压缩算法。

● struct zram_table_entry *table。

zram 中用于管理一个 PAGE_SIZE 大小数据的单元为 zram slot，对应数据为 zram_table_entry。

zram slot/zram entry 可以与 swap entry 一一对应。

● 关键字段

handle：用于映射访问 zsmalloc 中存储的压缩后数据地址。

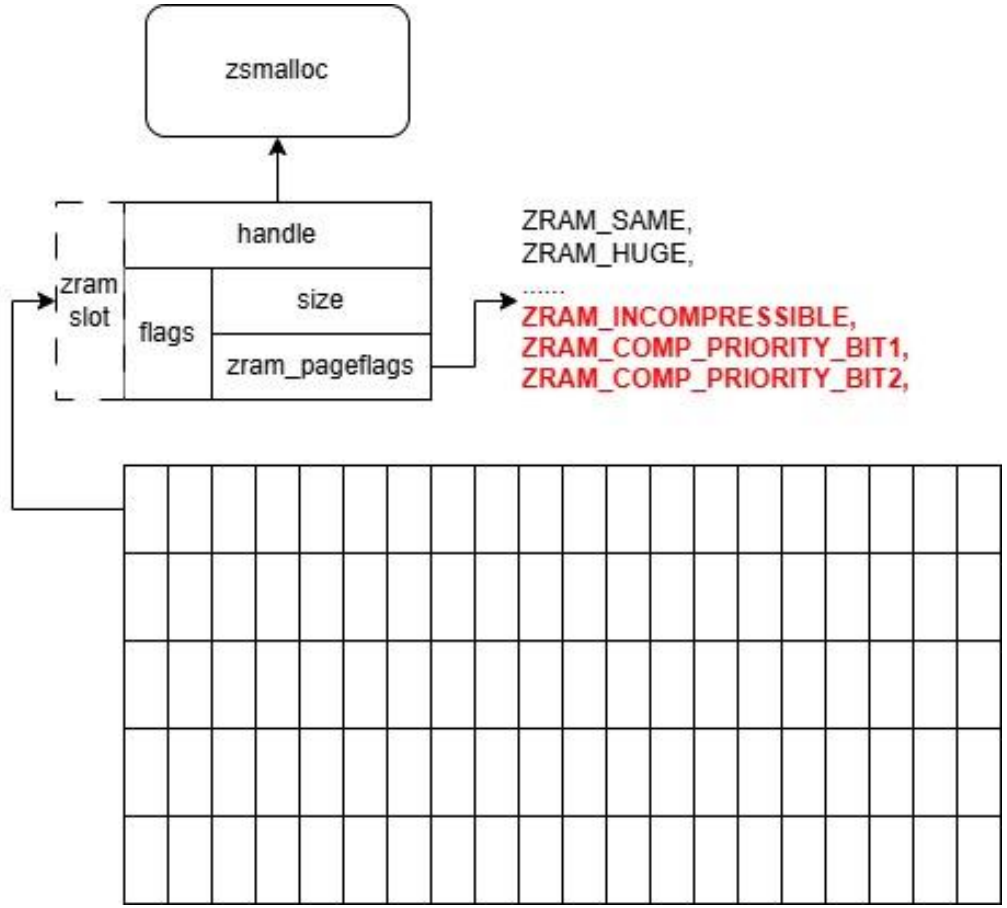
size：handle 映射的大小，近似压缩后大小。

zram_pageflags：按 bit 位标识该 slot状态。

● 增加 zram_pageflags

ZRAM_COMP_PRIORITY_BIT1/2：2个bit位(4种)标识当前 slot 使用了哪种压缩算法。

ZRAM_INCOMPRESSIBLE：尝试了所有的压缩算法，依旧无法腾出更多内存的 zram slot。



Part3：ZRAM_MULIT_COMP-使用

目标：让 zram 重新压缩指定数据，减少 zsmalloc 的使用量。关注压缩率但牺牲压缩/解压速度。

● echo “type=huge" > /sys/block/zram0/recompress

遍历 zram slot，根据输入的条件，将符合的 zram slot 先解压，再按顺序用新压缩算法重压缩，直到新数据的大小比原数据小，则更新 zram slot 中的压缩数据、大小及使用的压缩算法。

如果遍历过以上各种压缩算法均无法压缩到更小，则标识为不可压缩的slot(ZRAM_INCOMPRESSIBLE)。

a. 配合 zram writeback 特性（将zram内数据回收到存储，进一步释放内存），可优先回收到存储。

b. 下次 recompress 遍历时直接跳过

● 可选配置

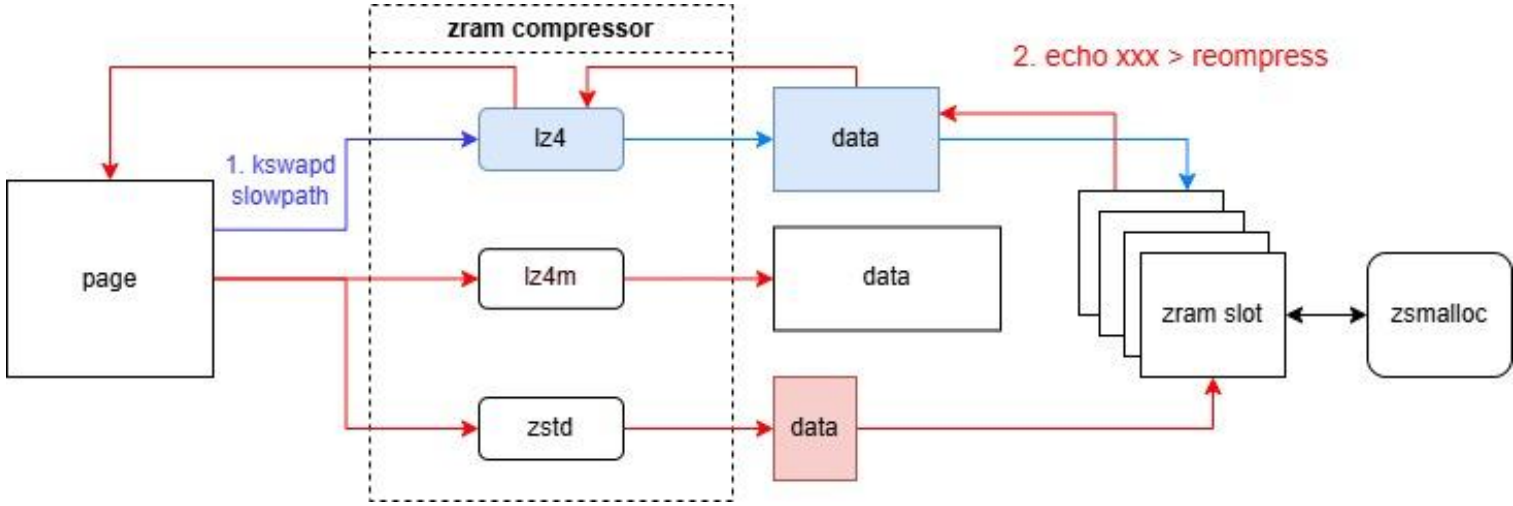
type=idle/huge/huge_idle，类型

max_pages=1024，限制每次重压缩数量

threshold=888，临界值，选择大小大于该值的slot

重压缩到小于该值

algo=zstd，priority=1，使用指定优先级或指定算法重压缩



Part3：ZRAM_MULIT_COMP-问题

优势：

让 zram 可支持同时使用多种压缩算法。不同算法各有优劣，从而得以在不同场景发挥各自优势。

问题：

● 速度太慢不可接受的。

如作者建议的 zstd，在 Android 低端机器小核低频测试速度如下。回收/读回速度过慢，导致进 recompress 回收时间长，读回时卡顿掉帧。

There are several use-case for this functionality:

- huge pages re-compression: zstd or deflate can successfully compress huge pages (~50% of huge pages on my synthetic ChromeOS tests), IOW pages that lzo was not able to compress.

```
[redacted] # cat /sys/block/zram0/comp_algorithm
lzo lzo-rle lz4 [zstd] lz4m
[redacted] # taskset 01 ./zram-perf-path 1 ./zram-dump 128M
direct=1, in path=./zram-dump, len=134217728
write: 7.00163MB/s
read: 31.8785MB/s
```

● 压缩算法的选择，仅考虑压缩率，未考虑压缩解压速度，不够完善。

Part3：多压缩算法使用

目标：解决重载场景启动更耗时问题，重载时更快腾出内存，且无负面影响。

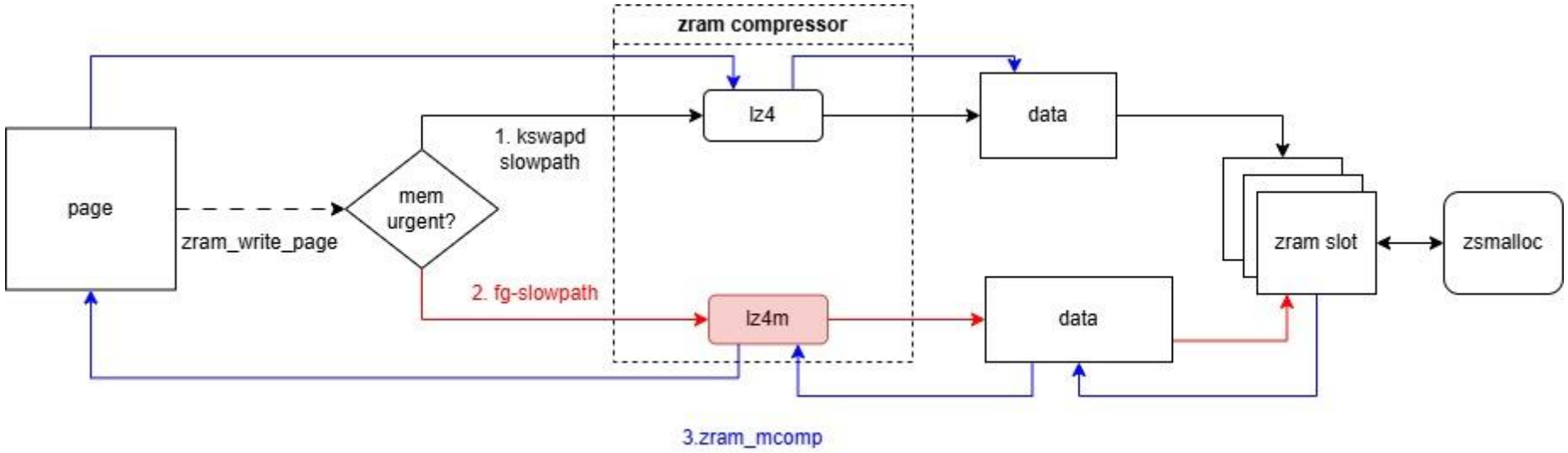
方案：以腾内存速度为标准做第二压缩算法的选择，配合 recompress 保证系统总体压缩率最优。

配置、管理：

lz4 -> comp_algorithm 主压缩算法，lz4m-> recomp_algorithm 第二压缩算法

使用：

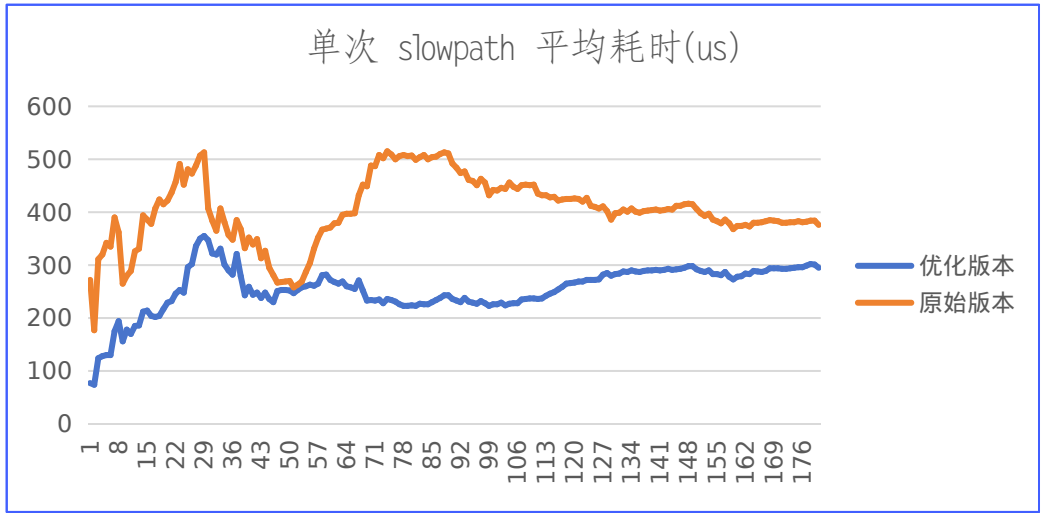
- **场景关联**
  - kswapd 与绝大部分 slowpath 保持用较平衡且压缩率较高的 lz4 压缩。 → 保障压缩率、保活
  - 前台关键进程进 slowpath，使用 lz4m 压缩，可更快腾出内存。→ 优化 slowpath，减少启动耗时和波动
- **精准重压**：增加内核线程，在闲时主动遍历使用 lz4m 的 slot 做 recompress，保持系统总体压缩率最优。 → 保障压缩率、保活



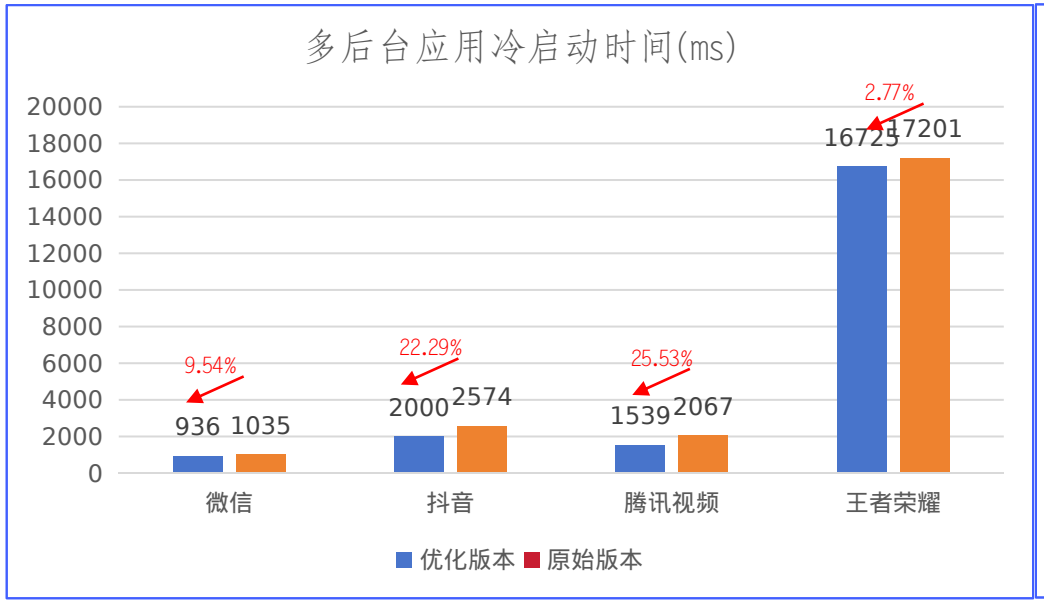
Part4：效果及展望

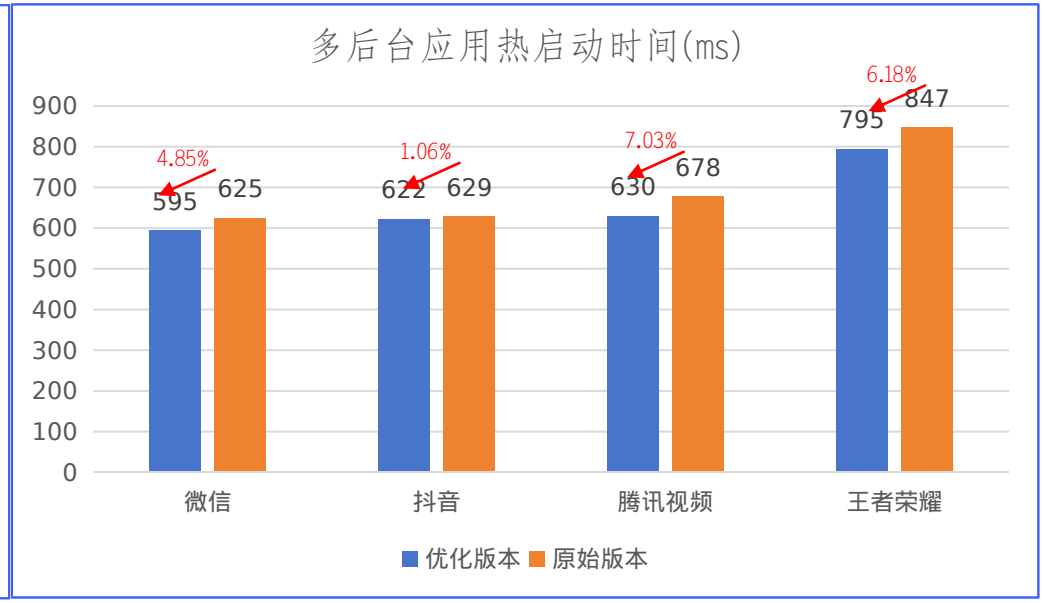
Part4：效果

● **技术指标：** 单次 slowpath 耗时减少 25%。



● **用户场景：** 重载多后台场景应用启动耗时优化 2%-20%





Part4：展望

- **压缩算法：**使用场景扩展，非内存回收数据，做特定优化和使用。
- ZRAM：cpu 侧算法优化 -> 异构方向

THANKS