Contents

List of Figures :

Abstract

No embedded system is possible without a memory inside as memory is important part of any system. So managing the memory is a crucial part of work in the System. That is why memory management is the most complex part of operating system. Whenever an operating system is decided for any devices the device memory and OS capabilities are matched. Choice of Android for the Mobile Phones give best combination of under lying hardware and software capabilities. In lined with the choice of OS, Android provides improved Memory Management than Linux. Android also provides the in build policy for handling with the low memory scenarios. But as the performance has no limit to achieve and daily user requirements for faster and bulky software needs further optimizations. Here in this thesis our aim is to focus on the RAM availability for the System. More system RAM means improved sluggish behavior with enhanced degree of multi programming and responsiveness. We studied various method of in kernel compression to achieve goal of improved RAM. These methods are Com-Cache, zSWAP, KSM. These methods are not in the proper shape to use with Android Operating System and Android Low Memory Policy situation. After deep analysis here to enhance the RAM availability we need to change the Android Low Memory Policy and modified com-cache and KSM implementations to suite the Mobile Phones. KSM utilizes the CPU heavily and drain out battery very fast and Com-Cache is not compatible with the LMK in its raw form. These techniques are modified and also changed Android Low memory Policy and these results in batter RAM availability to the system.

Chapter 1

Introduction

When we design an embedded system memory is integral part of it. There are several options for memory like NOR,NAND, eMMC ,PCM. Various types of RAM (DDR1,DDR2, stacked, non stacked). Every piece of software stored in some kind of memory. When a device is powered on if we are using NAND or PCM based devices software is first brought to the RAM and from there it starts executing. If NOR memory is used software can be directly executed form NOR without bringing it to the RAM unlike in NAND and PCM case. This is because of the XIP property of NOR memory. CPU is linearly addressable and NAND/eMMC etc are block devices so only block chunk is first read and then bring to RAM form there rest of software begin to execute. All most all the smart phones are NAND/eMMC based so RAM cannot be ignored. So if we have more RAM than more programs can be accessible without read from NAND/eMMC and makes the accessibility of application faster. Moreover RAM is faster than other FLASH memories.

Motivation of the Work

Selection of memory in case of Embedded system is most important as it impacts the BOM as well as performance. Before smart phones most of the phones were NOR based phones, to run all most all the available software 256MB of NOR with 512KB/1MB/2MB of RAM was sufficient based on the requirements. But NOR cannot be used with the smart phones as Smart phone software quite complex and take more space as they are converging to TV technology, cloud computing and many more paradigm. As NOR is costlier than NAND so it is no more the cost effective solution. NAND is also being replaced by the eMMC, the problem is the File Transltion Layer required and if pure NAND is used all the operations are performed in the Software, eMMC gives a batter solution with inbuilt controller which is takes care of FTL operations. So all the Smart Phones of today’s are using the eMMC over SDIO interface. So when these technologies are used RAM usage is increased because if linear addressing scheme of the CPU.

The motivation of this work is to avail as much as possible RAM to the system so that new technologies for the Smart phones never lacks it. This is not 100% possible as software requirements for the main memory is always increasing. Other than the hardware factors as described above design of the operating system is another consideration. Android design is un doubtfully RAM Hungary. Android is so far used in hand held devices specifically for the Mobile Phones, which characterizes as the small screen (5”,7”, 10”) size seems sufficiently large when compared with old phones but as mobile is converging as TV or desktop so still very small. When we are using the phone and switch form one action to other says from watching moving to composing the SMS, media player is not killed but it goes to the background. Also form SMS to if one switch to reading the paper so SMS typing window is not destroyed. But being a short screen all these activities are not visible on the screen but remains in the background consuming the RAM. The phones with faster response always try to keep maximum number of applications in the Background but this is again limited by the availability of RAM. The Linux and Android already provided few techniques to take care of RAM availability pressure but still batter techniques can be developed.

Related Work

Many researchers have proposed the techniques for effective usage of RAM at the application level and at the System level. We are not considering the application based techniques as there life span ends with the popularity and day to day up gradation, target deployment device. Like in case of database designing there are various techniques for Oracle cannot be used for the Mobile. Different Video Players are designed for different requirements of the devices. So our focus was on understanding the System Level Optimization Techniques.

In the system side following techniques has been proposed

Com Cache: in-memory compressed swapping :

The concept of memory compression compress comparatively unused pages and store in main memory. This is simple concept and used from quite long in the Operating Systems. Here philosophy is to use compression, and reduce expensive disk I/O operation. This proves much effective than swapping pages to secondary storage. When a page is required again, it is passef through decompression and send back, which is, once again, much faster than going than swap. This idea is under implementation for Linux is with the name as the compcache project. Here a virtual device is created call it ramz swap which acts as swap disk. Pages are swapped to ramz swap disk after compression and stored in memory itself. The goal of the project is not just performance on swap less setups, it allows running applications that would otherwise simply fail due to lack of memory[1].

zswap compressed swap cache :

Swapping means performance degradations as it involve the I/O operation form fast device to slow device and vice-versa. The I/O latency between RAM and swap, even with fastest SSD, is of the order of magnitude four. The throughput gap is of orders of magnitude two. In addition to the speed, storage on which a swap area resides is now a days can be more shared and virtualized environment, which can cause furtherl I/O latency and non deterministic performance. The zswap goal is to mitigate such undesirable effects of swapping through I/O activity reduction. “zswap” is a write-behind, lightweight cache compressed for swap pages. It selects pages which are being swapped out and it tries to compress them into a dynamically allocated RAM-based memory pool. If this process is successful, the write operation to the swap device is delayed and in most cases, can be avoided completely. It results in significant I/O reduction and performance improvements for systems that has swapping capabilities[8].

Increasing Memory Density Using KSM:

In case when we are having the same host an on that different virtual machines are running, with the possibility that they are handling the same software and data the possibility of RAM duplication is very high. KSM is implemented as the Linux kernel loadable module this allow sharing of anonymous memory across different virtual machines to be shared. KVM is treated as the another process in the Linux system not different a different process. So mmu notified and other things remains same Guest physical memory is allocated as regular Linux anonymous memory mappings. KSM scope is not limited to the virtual machines[9].

The main task of KSM is to identify same pages from the system. For this it uses two red-black trees, one is called the stable tree the other is called the unstable tree. The stable one contains past shared pages and not frequently changed pages. The unstable red-black tree holds pages that are still not shared but are under the supervision of KSM.

Problem Statement

Research Gap:

The LMK in its present form kills the processes based on the algorithm of process priority. The killing order is Empty/Background App, Content Provider , Hidden App, Secondary Server, Visible App, Foreground App. No further options are explored in order to save the processes form being killed. if these processes are killed and in very next moment if processes are needed again they are started again so phone exhibit the sluggish behavior. The number of times the LMK is called is potential parameter for the sluggish behavior of the phone. The draw backs of the frequent LMK and OOM execution are

- Phone exhibits the sluggish behavior.
- During process loading the memory controller is accessed frequently this cause more current consumption
- Loading the complete process takes more time than bringing the process form back ground to foreground.

If memory pressure is not controlled by the LMK then OOM is inevitable. The problems with OOM are more severe as

- OOM process selection is not controllable form the Framework
- If a system process is selected for the killing then system is going to hang surely
- It does not care for the fore ground and background processes

The ultimate goal of the thesis is to control the "memory pressure" on the system by increasing the available RAM to the system. As the in current Android System the LMK algorithm is used to counter the memory pressure. *Here in this thesis LMK algorithms is modified and integrated with KSM and Com-Cache techniques and collectively gives the improved results to handle memory pressure*. The KSM and com cache in the present form are not applicable on the Mobile phones because of the

- High CPU usage resulting to the battery draining up fast
- Battery is one of the criteria for the Smart Phone
- High CPU usage also degrades the overall performance of the system

The primary reason is the basic techniques used where each page is compared with rest of pages to find the similarity among the pages. Each page is 4K and the comparison method is memcmp, even we optimize this trivial function to extreme based on the platform used CPU usage is very high. So far it is not commercialized in the Mobile Phones indeed it good for saving the RAM.

Story with the com cache is batter for CPU usage but the Android Low Memory Policy and Com-Cache are not compatible to use effectively. The calling point of Com-Cache should be in control of LMK and in which thread the compression and decompression is to be done so that rest of system is not impacted. Timing is also major concern. The important point still not answered in any research is relevant to in kernel compression does not answer reducing the LMK frequency count.

if we can improve LMK algorithm this can result to the batter availability of RAM with minimum impact on performance and degree of multiprogramming. Therefore problem statement is :

Statement:
"Proposing the modified Android Low memory handling policy to increase the RAM availability to the system with minimum impact of the performance without compromising degree of multiprogramming, and stability of the system."

Goals of Thesis

Handling the memory pressure is one of the tedious takes of any Embedded System. This become more complex with ever increasing demand of quick responsive behavior form the system. Linux uses OOM killer to cope up memory pressure but this is not suitable for android so Android uses LMK over OOM. The standard algorithms for the Android is merely based on the Android Process Management. This can be further improved as attempted in thesis with modified in kernel compression techniques such as KSM and Com Cache.

Major Goals:

1. Studied various methods of in- kernel compression techniques such as com cache, zwap, KSM, CMA in Linux.
2. To propose new LMK Policy to enhance RAM availability for Android Mobile Phone
3. Modify Com Cache so that it works with LMK of the Android
4. Modify KSM so that it can be used for the Android Mobile Phones and integration with LMK

Under New LMK Policy processes and pages are identified for compression when the system is in idle state or the system is under memory pressure. Even after compression memory pressure is not reduced background processes are identified and they killed to make room for new processes.

Com Cache is in kernel compression technique. Com Cache is modified with the Android System as the loadable module. Once LMK has identified the pages for compression com cache do the compression.

Input to the KSM is modified. Instead of applying KSM on all the processes, it is applied only on the background processes and processes with same ancestors. As the pages of these processes are most likely identical.

Chapter 2

Related Work and Background Work

2.1 Android Memory management

Android is the open platform and best for developing apps. More and more users prefr Android for the development of various applications. Memory requirements for the various apps are different some apps such as such as game, video player and so on, requires large memory, so the phone becomes slow with usage. The **Figure <2-1>** shows the Android phone memory requirements for different applications, and the **Figure <2-2>** shows the memory usage of most frequently used applications

Model	Total memory	Memory usage for app
I559	384 MB	286 MB
I579	384 MB	286 MB
I9000	512 MB	339 MB
I9001	512 MB	352 MB
I9100	1 GB	835 MB

Figure 2-1 Samsung Android Mobile Phone Memory

App name	Memory used
Phone	**23.4 MB**
UCweb	**9.9 MB**
Angry birds	**73 MB**
Plants vs. Zombies	**76 MB**
Angry bots	**75 MB**
Input method	**19.4 MB**
Camera	**8 MB**
Google map	**12.7 MB**
Email	**9.9 MB**
Gmail	**9.5 MB**
Market	**20.5 MB**
Music Player	**19.6 MB**
Video Player (720 * 480)	**23.2 MB**

Figure 2-2 Main Stream Application Memory Usage

Android memory management design is different from the traditional operating systems: On exiting an application it will not free the memory allocated to it. When system is under memory pressure then application memory of not in use application will be freed. So as we keep on using the phone for long time the count of such unused application will keep on increasing ultimately less memory will be available. In case if we now attempt to start a large memory consumption application in the low memory status, it will take long time to start the application. The solution in such a low memory scenarios is to be founded, Android is based on Linux kernel, it follows most of Linux kernel memory management[2].

2.1.1 How Linux manage memory

Linux is virtual memory based operating system. Generally linux machines has small amount of RAM and Large Virtual Address space. Linux supports both hardware and software mechanism to make sure those programs can execute transparently without knowing the fact that small physical memory is used. Each process in the Linux has its own unique virtual address space. These virtual address spaces are isolated from each other, a process running one application cannot access another application. The basic unit of memory is page. The relationship between physical addresses and virtual addresses is given by page tables, and if a virtual address is referenced and there is no corresponding physical address, page fault occurs, this is handled by operating system. Hardware allows fixed page sizes[1].

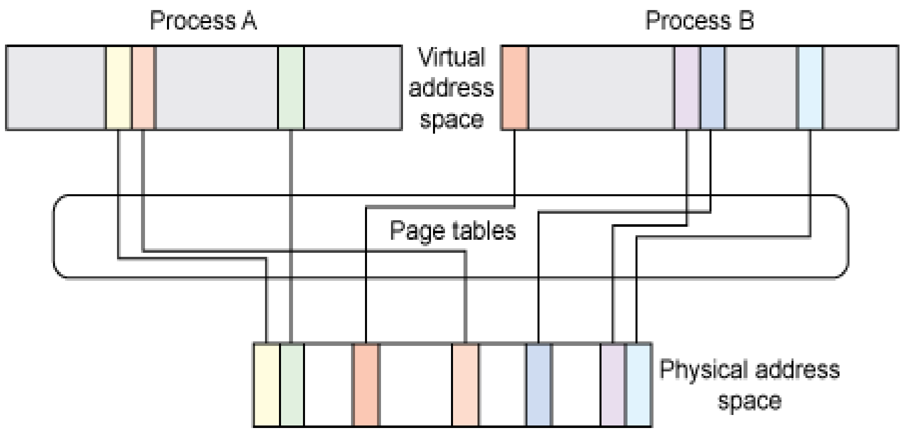


Figure 2-3 Virtual Address Vs. Physical Address

Accessing the code or data every time for the secondary memory is not efficient. Linux kernel uses caches concept to enhance system performance. Caches are termed here keeping the data in the RAM memory. From slow devices data is read and hold in RAM for short time, data still remains in RAM even if process is not active. So next attempt if process accesses the data, it is read directly from the RAM, without accessing the slow block device. When the system memory is under pressure first cache memory is freed slowly to make room for other processes to be loaded. SWAP increase the effective size of RAM so the total memory at disposal of the system is sum of RAM plus SWAP space. SWAP is used as extended RAM so that the effective size of memory grows with defining the SWAP space. The kernel will write the contents of a unused or inactive memory to the swap space making room to load other process. When the original contents are needed again, they are read back into memory. When in low memory case, if kernel needs more memory to start a process, but there is no swap space to swap in pages and no caches can be shrunk. In this situation, the higher application code instructs the OOM killer to kill the un-important process in the system to obtain large number of memory pages to load other processes. Refer to Figure<1-2>, the mechanism of swap, cache shrinkers and OOM killer will be start in sequence according to seriousness of low memory[4].

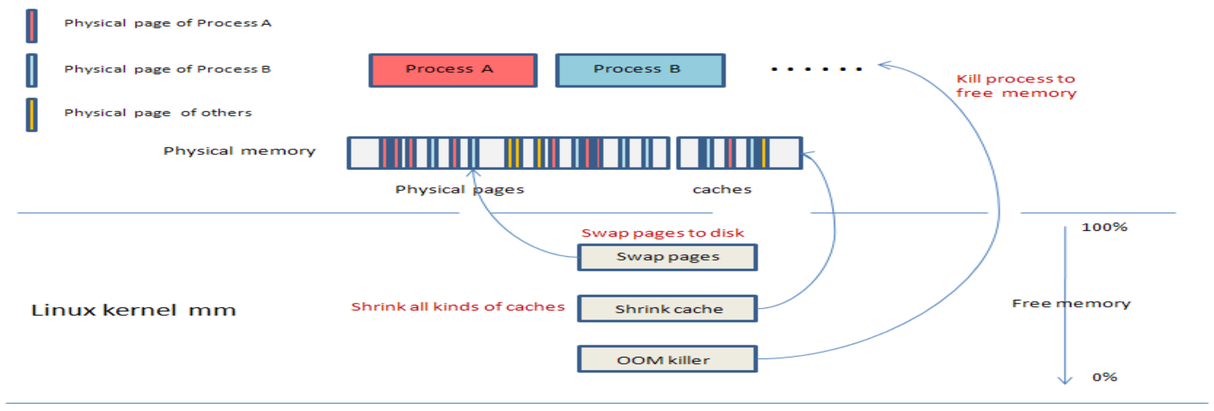


Figure 2-4 Low Memory in Linux Kernel

2.1.2 How Android Manages Memory

Android is based on Linux kernel; it uses the same Linux memory management, with little modification:

- In Android phone, there is no swap space. The phone’s storage flash or eMMC card don’t read/write very frequently, Android does not support swap feature.

- Android supports a kernel diver named Low Memory Killer (LMK) other than OOM killer that is standard Linux feature, when the system memory is lower, it kills less important application to free memory.

Android process management is different than Linux, unlike other operating system in Android if user switches to another Android application, the process is not killed but left in memory as background process. So it will be an empty application or background application, it just sitting idle, does not use any CPU, battery, or network capacity. This design is very effective optimization so next time when user switches back to that application, it will be loaded immediately, without reloading the resources again. But there is situation when lots of applications changed their status to background app, the system will be out of memory. Here system requirement is to kill some applications to make room for new applications, but selection of the applications to be killed require some intelligent so LMK helps. The use of traditional Out of Memory (OOM) to kill the process has some drawbacks[4]. There are some points about this:

When system lacks memory seriously then OOM killer is triggered. LMK is called far early when memory pressure increases the thresh hold. The LMK is registered in cache shrinker list, this is the list which reduces the cache when system is under memory pressure. LMK works with the cache srinker.

- OOM killer cannot be controlled form the user space, there is no way to importance of Application can be registered with the System, so it is not guaranteed that OOM killer will kill least important application. Android framework divides the Android applications to various categories and LMK is aware of various categories of lass of android application, and LMK follows the order of categories of applications starting from empty applications, content providers, background apps, fore ground apps[5].

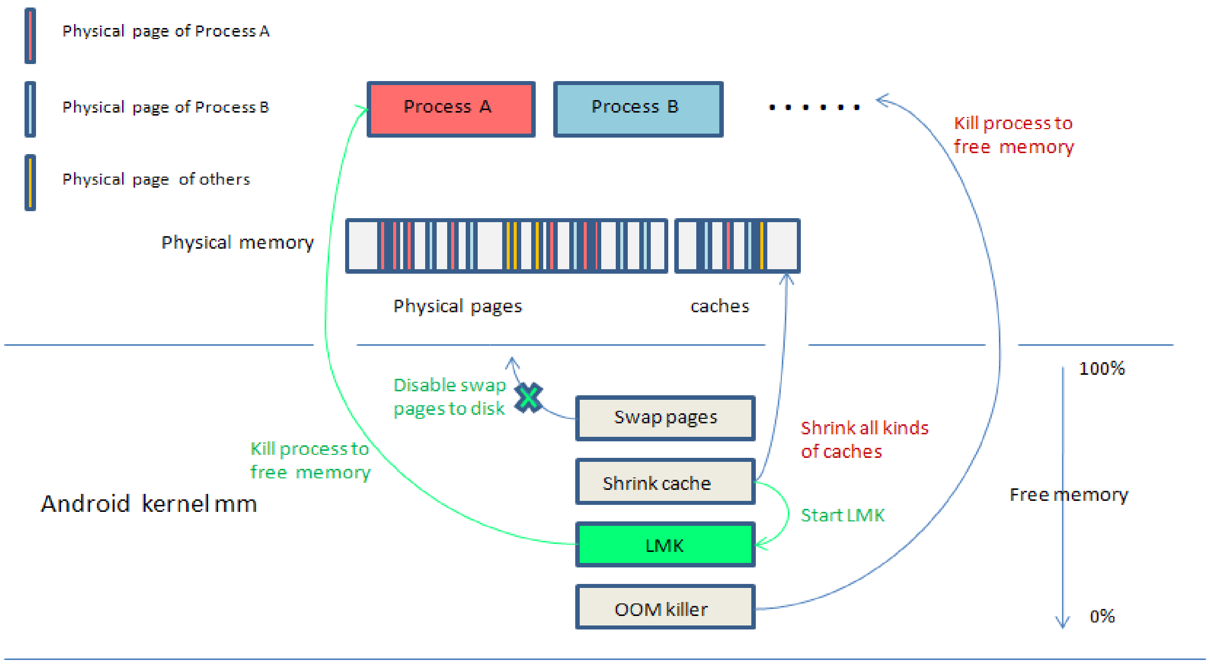

Physical page of Process A
Physical page of Process B
Physical page of others
Process A
Process B
Kill process to free memory
Physical memory
Physical pages
caches
100%
Disable swap pages to disk
Swap pages
Shrink all kinds of caches
Kill process to free memory
Shrink cache
Android kernel mm
Start LMK
Free memory
LMK
OOM killer
0%

Figure 2-5 Low Memory in Android

2.2.3 Optimization of Android memory management

With Android memory management, we can run applications as much as possible till the system fails to allocate pages. If we can increase main memory we can run more applications, it is good solution, and surely the mobile have more memory so run efficiently. Now, just think of SWAP feature, it is one of way to increase the physical memory size. Flash cannot be used in Mobile phone because of the NAND properties. Comp-cache/zRAM is a good solution of embedded system memory management. Comp-cache creates RAM based block device (named Ramzswap) which acts as swap disk. Pages are swapped to this RAM area are compressed and stored .Compressed pages in RAM increases capacity. This allows many applications remain in memory.

For Android to enable SWAP feature, LMK policy has to be defined working with swap, when system is in low memory situation, first it swap out pages ; if swap pages cannot free enough memory, Low Memory Killer (LMK) will be called.

2.2 SWAP

SWAP is basic concept of the most of modern Operating System. In SWAP implementation a small portion of the Hard-Disk is reserved. This space is used by the Memory Management Code to store the ready pages. RAM is the faster memory so Operating system will try to keep most of the data in the RAM. When the RAM is full the Memory Manager will move inactive pages, ie pages not in used for long time to the hard disk, freeing up RAM for the active processes. If any of this page from the hard disk needs to be accessed again, it will move back into RAM, and another inactive page from the RAM is identified and it is moved onto the hard disk ('swapped'). The SD cards and FLAH are considerably slower than physical RAM, so when some page needs to be swapped, there is a noticeable performance hit[2].

As swapping is the slow process, identification of pages to be swapped is major concern. Kernel requires an efficient algorithm to find the pages for system performance.

2.2.1 SWAP Code

As mentioned above SWAP policy should not lower the system performance. After moving the pages to SWAP page frames are freed for other processes, so it also called called page reclaim. SWAP implementation is one of the most complex part of the kernel. - “try_to_free_pages” This API is invoked when kernel finds extreme shortage of memory during execution of a process. It scans all pages active the current Memory Zone and frees least frequently used.

- There is a background thread known as “Kswapd” the job of this thread it to records memory usage at frequent intervals and identifies memory shortage. This data is used as input to swap out pages as a precaution before the kernel jumps into the situation of not sufficient memory. Source code is explained in the form of steps:

Step1: “shrink_zone” is used for removing the pages from the memory. This method does two things: Its role is to maintain a balance between of active and inactive pages in a zone by transferring pages between the inactive and active lists with the use of shrink_active_list. shrink_zone decides number of pages of a zone are to be swapped out and decides which pages.

shrink_active_list is a supporting function called by the kernel to transfer pages between the inactive page and active lists. The function is given input parameter as the number of pages to be transferred between the active and inactive lists and then tries to select the active pages least used. shrink_active_list is responsible for deciding which pages are swapped subsequently out and which will remained. Here is policy part of page selection is implemented.

Step2:”shrink_inactive_list” It removes inactive pages from the inactive list from a specific zone and releases them to shrink_page_list, these pages are again reclaimed by specific requests to the backing stores to write data back free space in RAM. If for some unspecified reason, pages are not written back shrink_inactive_list must put them back on the list of inactive or active pages.

2.4 Comcache

Comp-cache (compressed caching) is a technique to set aside some portion of the memory to reserve where compressed pages will be stored during swap out. Comp-cache do compression-decompression and store and retrieves the pages from RAM You effectively get more RAM from the compression[7].

To implement Comp-cache, Linux kernel will create RAM based block device acting as swap area (named Ramzswap). Developer can define the size of Ramzswap (the default value is 25% of total physical internal memory), this defined size of Ramzswap is just a upper limit for the swap area. It does not mean the size of Ramzswap is fixed, on the contrary, it is flexible:

For example, the total internal memory is 800MB, define 200MB as the size of Ramzswap,

- At initialization, system does not allocate all the memory (200MB) for Ramzswap as its pre-defined size (only some initialization parameter, data structure are allocated); at this moment, system can allocate nearly all the 800MB internal memory to those active processes.
- During Comp-cache working, system will allocate free physical pages to store those compressed swap pages, according to Comp-cache requested. (but the total size cannot exceed 200MB, as we pre-defined the 200MB as the size of Ramzswap, it is the upper limit of Ramzswap)
- When Comp-cache de-compress and swap out pages to internal memory, these pre-occupied pages in Ramzswap will be return to Linux kernel as free pages, and can be allocated to other active processes, when they request free pages.
- When Ramzswap size reaches its upper limit (here, we defined as 200MB), Comp-cache will not swap any more pages to Ramzswap.

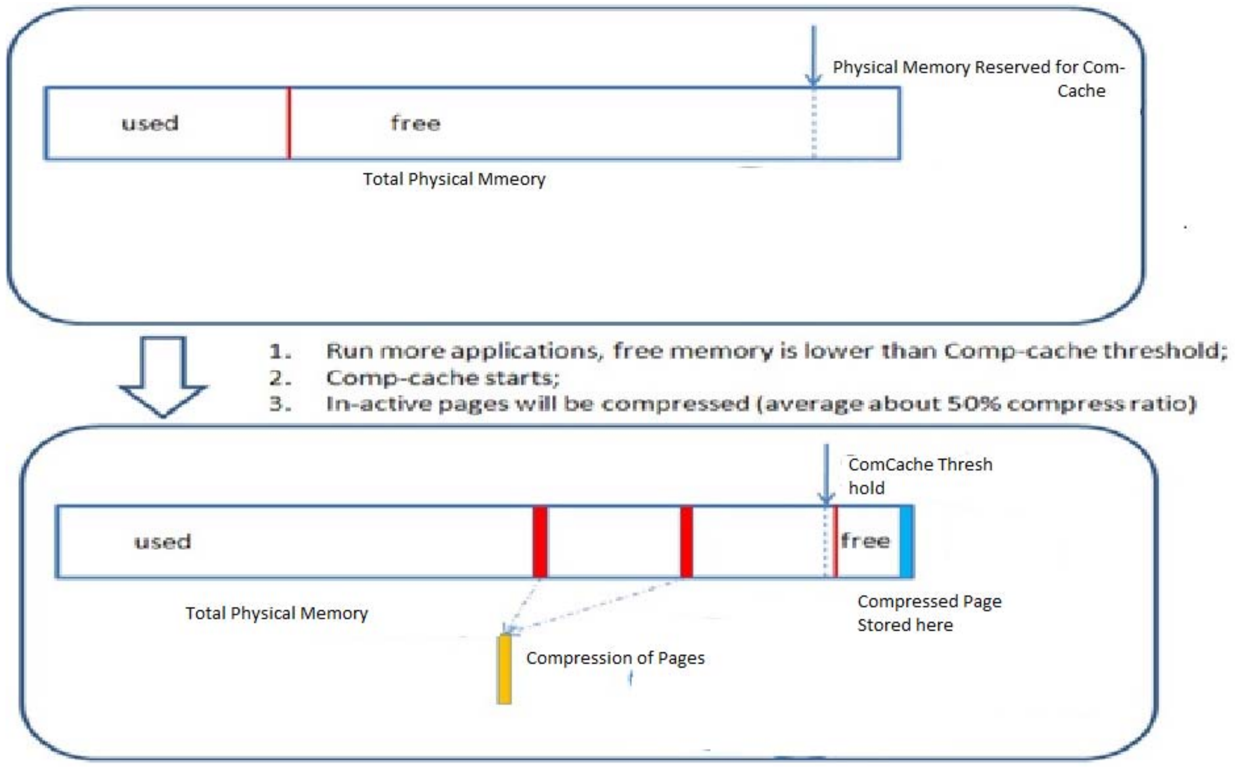

Physical Memory Reserved for Com-Cache
used
free
Total Physical Mmeory
1. Run more applications, free memory is lower than Comp-cache threshold;
2. Comp-cache starts;
3. In-active pages will be compressed (average about 50% compress ratio)
ComCache Thresh hold
used
free
Total Physical Memory
Compressed Page Stored here
Compression of Pages

Figure 0-6 Swap Code Architecture

Note:

- The actual “used area”, “free area” is not continuous as this picture described. In actual physical memory, they are organized as physical frame page (each for 4KB); those un-used physical pages are “free area”, they locate discontinuous, those used physical pages are “used area”, they also locate discontinuous. Here we just make it easy to understand for memory area changing status.

-“Ramzswap” is a logic name. All those physical memory that used for storing compressed pages are named as “Ramzswap”. These pages are also discontinuous, and they are part of the “used area”.

Following diagram **<Figure 2-7>** is describing from a real physical frame page allocation, it will help for understanding the real memory distribution situation:

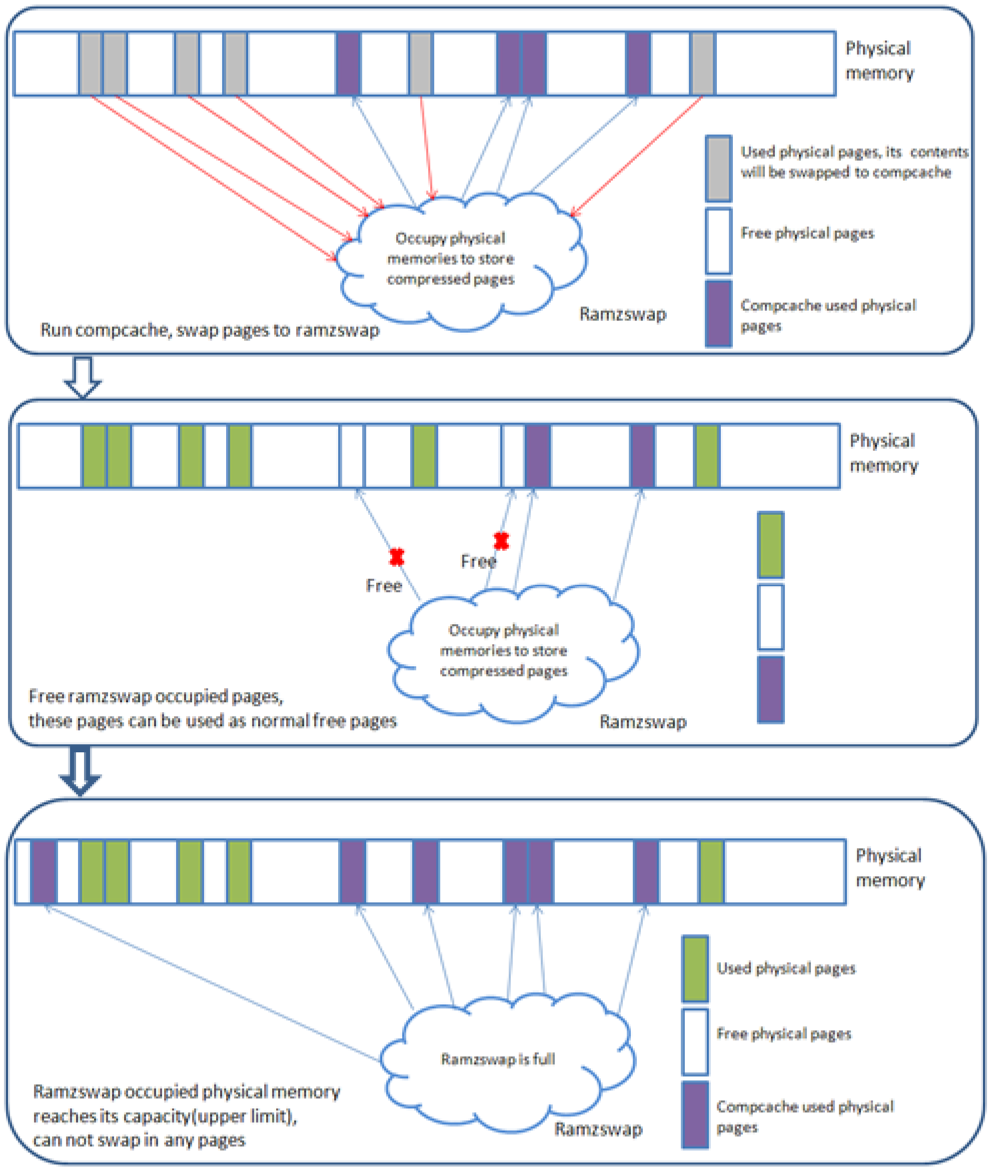

Physical memory
Used physical pages, its contents will be swapped to compcache
Free physical pages
Compcache used physical pages
Occupy physical memories to store compressed pages
Ramzswap
Run compcache, swap pages to ramzswap
Physical memory
Free
Free
Occupy physical memories to store compressed pages
Free ramzswap occupied pages, these pages can be used as normal free pages
Ramzswap
Physical memory
Used physical pages
Free physical pages
Compcache used physical pages
Ramzswap is full
Ramzswap occupied physical memory reaches its capacity(upper limit), can not swap in any pages
Ramzswap

Figure 0-7 Com Cache Working scenario 2

As the Figure <2-8> show, when all the system memory is occupied by Application 1, 2 and 3, only 2 pages are free, (at this time, part of App 3's pages are compressed and stay in Ramzswap) meanwhile Application 4 want to start, it request 4 pages. The system have no enough memory , then the kernel will swap out some inactive pages to get free memory, so some pages need to be swapped to Ramzswap. The Comp-cache will compress the pages before save them, for example , if swap out 4 pages, it only need 2 pages size to store in Ramzswap:

- System has 3 free pages.

- 2 pages of App 2 are compressed to 1 page, and stored to “page A”, then “page A” is part of Ramzswap (assume the compress ratio as 50%).

- Now we have 4 free pages, they are allocated to App 4.

- App 3 occupies page 1/2/3/4/5, among them, page 2/3/4/5 is belong to Ramzswap; When App 3 exits, all of the page 1/2/3/4/5 will be free, and return to system (page 2/3/4/5 are not belong to Ramzswap anymore); all of these pages can be used by others.

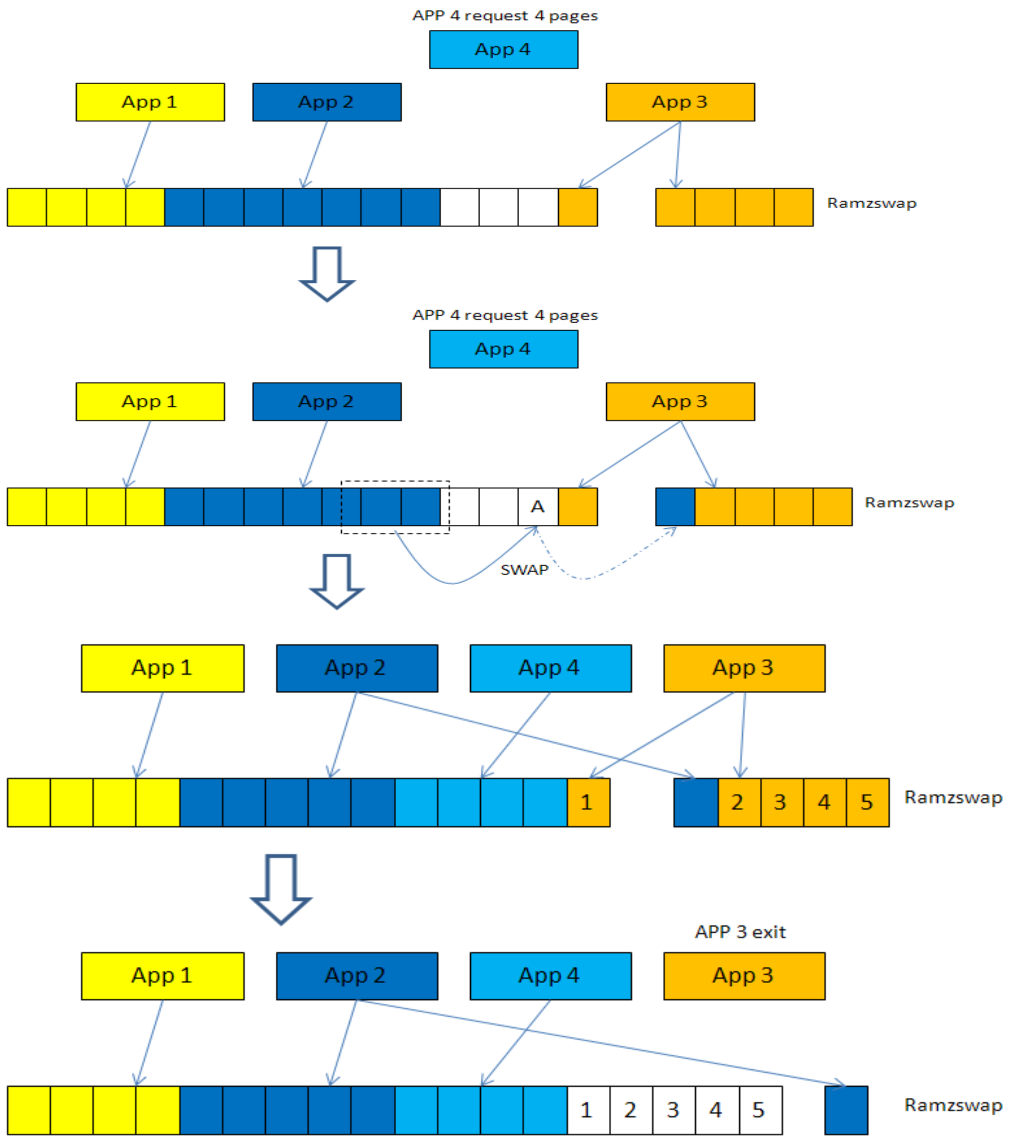


Figure 2-8 Comp-cache Working Scenario 3

2.4.1 Compression Ratio and Time cost of comp-cache

As the I/O from RAM is much more faster than hard disk, Comp-cache is faster than hard swap disk even if compress/decompress take some times. The Figure <2-9> show test data of comp-cache. We can see the average times of write one page to comp-cache is less than 300 us, average times of read one page from comp-cache is less than 150 us, and the compress ratio is good, it is less than 50%. Take last line in table for example, compress 38068 pages(152M) at 45.2% compress ratio, we can save 83M.

Item 1(Write pages) show the numbers of page that are written to comp-cache.
Item 2(Write times) show the overall times of writing such number of pages to comp-cache.
Item 3(Write a page times) show the average time of writing one page to comp-cache.
Item 4(Compress ratio) show the ratio of the total number of bits after compression to the total number of bits before compression. (the compress performance is better than 50%)

Write Pages	Write time (us)	Write a page time (us)	Compress ratio	Read Pages	Read time (us)	Read a page time (us)
768	129480	169	33.9%	17	2025	119
1819	310007	170	32.9%	57	5284	93
2089	351905	168	33.4	621	42906	69
3756	551668	147	30.9	821	57583	70
4410	709815	161	34.7	1181	120811	102
7459	1424428	190	35.4	1205	123241	102
11218	2137130	190	38.4	1386	180097	130
18718	3410624	182	40.9	2035	236825	116
23710	4418848	186	39.3	7680	988406	128
27399	5513179	201	39.7	8648	1055153	122
30873	6503889	210	43.0	10275	1187398	115
33328	6903785	207	43.4	12050	1333126	110
38068	7995497	210	45.2	15257	1514065	99

Figure 2-9 Comp-cache Read/Write Pages Time Cost & Compress Ratio

Compress time (us)	NAND write time (us)	NAND read time (us)	Decompress time (us)
103.3	134.4	94.8	58.7
Swap out		Swap in	
237.7		155.5	

2.4.2 The work flow of comp-cache

zRAM/Comp-cache has three major modules: Xvmalloc,Ramzswap , and LZO. Com-Cache is the driver initializes and creates RAM block device it act as swap disk, it will handle block I/O control. Allocator driver : Xvmalloc, used by the kernel when it swap one page to the RAM disk, it allocates some space from the disk memory. LZO is decompress/compress library, when kernel swap pages to the RAM disks, it will compress before it store, and decompress the pages when read from RAM disk. Figure <2-10> show the comp-cache flow chart:

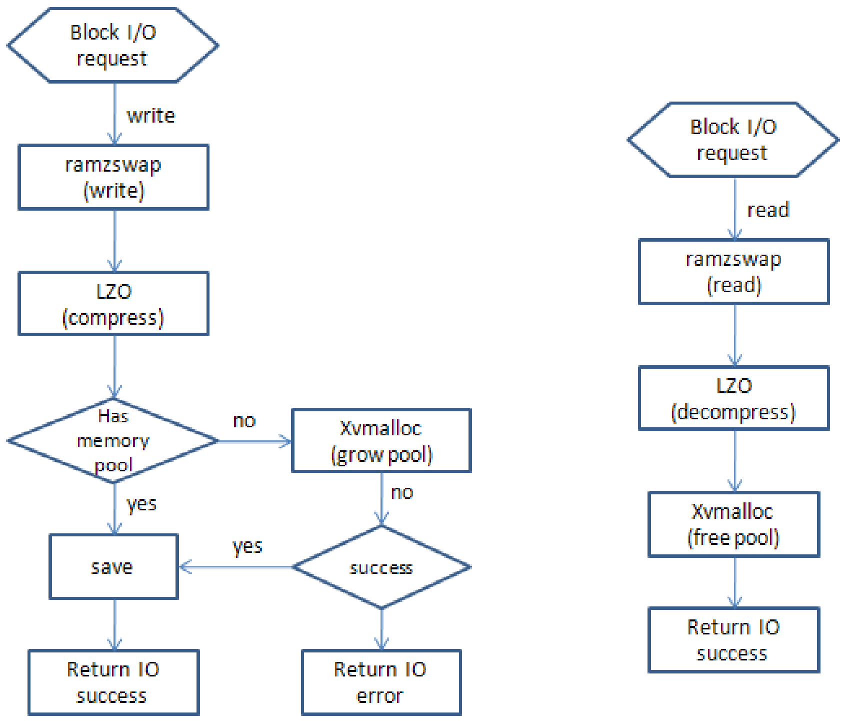


Figure 2-10 Comp-cache Working Flow

2.4.3 The comcache/ramz swap size verification

To verify the flexible size of Ramzswap, we tested on GT-I9103.

The total physical internal memory of G-I9103 is 1GB = 1000MB

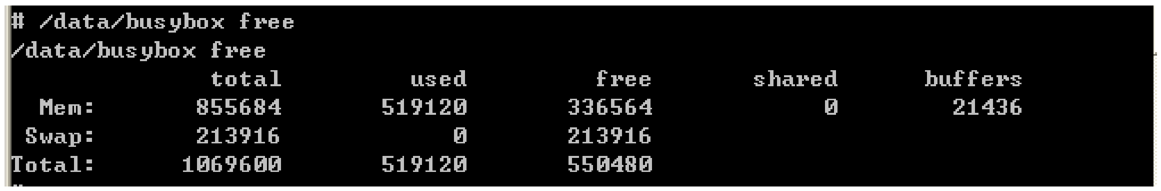
Including:

Linux kernel available memory: about 150MB

User space memory : about 850MB

Set aprox 25% of User space size as Ram zswap size upper limit.

In normal status: (Use "free" command in busy box)



```
# /data/busybox free
/data/busybox free
             total         used         free       shared      buffers
  Mem:      855684       519120       336564            0        21436
 Swap:      213916            0       213916
Total:     1069600       519120       550480
```

Figure 2-11 Memory Usage in Normal Status

Note:

The memory that occupied by Linux kernel is not calculated in.

"Mem" line: all the User space occupied memory is calculated;

"total column": total size of User space---- around 866MB

"used column": size of used memory in User space---- around 526MB

"free column": size of un-used memory in User space----around 346MB

"Swap" line: memory status of Ramz swap

Here, in "total column", the value "213916", only means the upper limit of Ram zswap.

It does not mean system has allocated "213916"B for Ram zswap. "Total" line: in "comp-cache" situation, this line has no meaning. Comp-cache is working on Linux SWAP mechanism; so, when we use “busybox/free”, it gives memory according to original Linux SWAP way; recall the SWAP diagram:

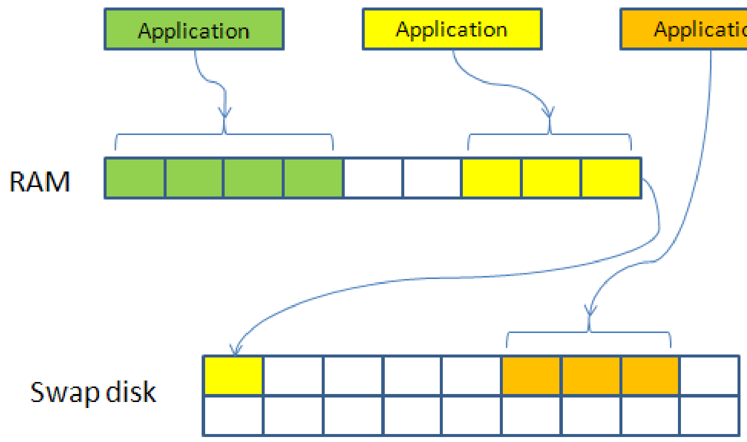


Figure 2-12 Linux Swap Mechanism Scenario

In original Linux SWAP mechanism, Swap area is in outside hard disk; when system initialization, system will allocate swap area memory in outside hard disk; Under this situation, the “Total line” indicate the total size of internal “User space” memory size and outside swap disk size. So, in “Total line”, we find it indicate “1069600” = “855684”(Mem/total) + “213916”(Swap/total). But actually, this value (“1069600”) is no meaning; as:

In Android system, there is no outside swap disk exist; only Ramzswap, which is not allocated pages at system initialization;

The value of “Swap/total”(“213916”) is only a upper limit for Ramzswap;

The “Total/total” value has exceed the real G-I9103 total internal physical memory size.

When more Applications are running, the free memory will be less and less; until the free memory is lower, that the system is in “slight memory shortage” status, the Comp-cache starts; system compress & swap in-active pages in LRU list to Ramzswap:

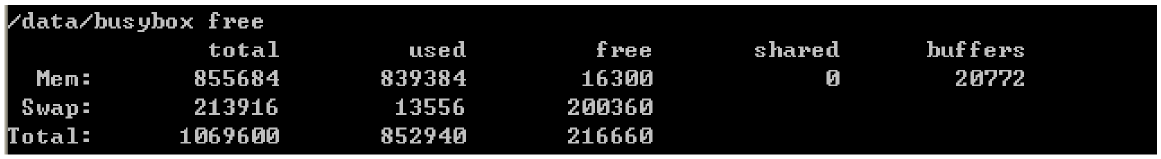

```
/data/busybox free
              total         used         free       shared      buffers
  Mem:       855684       839384        16300            0        20772
 Swap:       213916        13556       200360
Total:      1069600       852940       216660
```

Figure 2-13 Memory Usage After Comp-cache

Note:

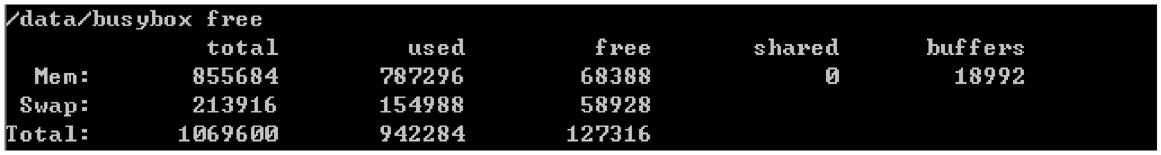
"Swap/used" means, current Ramz swap size is "13256"B; this value is part of the "Mem/used" ("839300"B)

"Mem/free" means, currently, system has only "16256"B free memory

"Swap/free" means, currently, system can compress & swap pages into "200360"B; But ATTENTION: it does not mean system still has "200360"B free memory; it only indicates how many memory can be used as Ramzswap.

When any applications is exit from the system, system will free all allocated pages (including pages used as Ram zswap and internal memory) used by these applications. All of these pages will be returned to kernel, and now can be used as normal memory.

Before application exit:

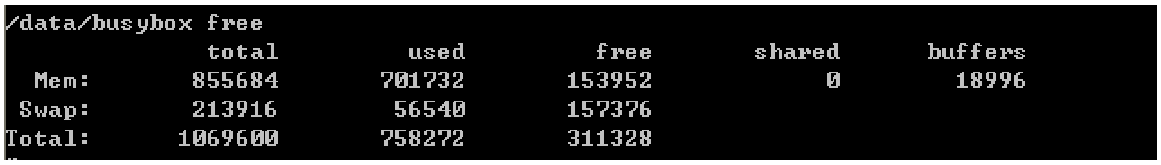


```
/data/busybox free
              total         used         free       shared      buffers
  Mem:       855684       787296        68388            0        18992
 Swap:       213916       154988        58928
Total:      1069600       942284       127316
```

Figure 2-14 Memory Usage Before Application Exit

In "Swap/used", "154988"B is used as Ram zswap. "Mem/free" indicates there are "68388"B free memory.

After application exit:



```
/data/busybox free
              total         used         free       shared      buffers
  Mem:       855684       701732       153952            0        18996
 Swap:       213916        56540       157376
Total:      1069600       758272       311328
```

Figure 2-15 Memory Usage After Application Exit

In "Swap/used", "56540"B is used as Ram zswap, almost "100000"B memory are return to system; "Mem/free" is added to "153952"B, almost "100000"B free memory added; Means, the freed memory from Ramzswap, is return to the system free memory.

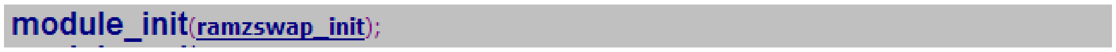
As a conclusion:

- During Compile, define the Ram zswap size as the upper limit;
- When system initialization, it will not allocate any physical memory to Ramzswap;
- During Comp-cache, the Ramzswap size will increase, according to how many in-active pages are compressed & swap;
- If the compressed pages are called by system again, they will be decompressed & swap out, those corresponding occupied pages in Ramzswap will also be free, and return to system; (Ramzswap size is decreased)
- When the application exits, the corresponding occupied pages in Ramzswap will be free and return to system. (Ramzswap size is decreased)

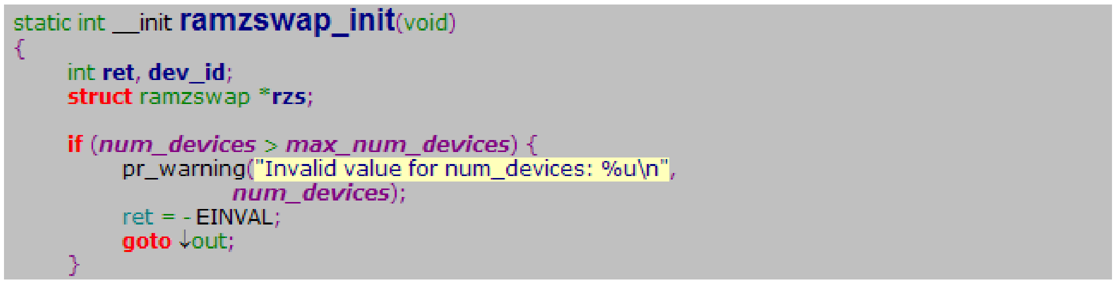
2.4.4 Comp-cache code

Step1: A virtual block device which cat as a swap disk is created. Pages moved to the created are compressed before it is stored in the memory.

Kernel/drivers/staging/ramzswap [12]:



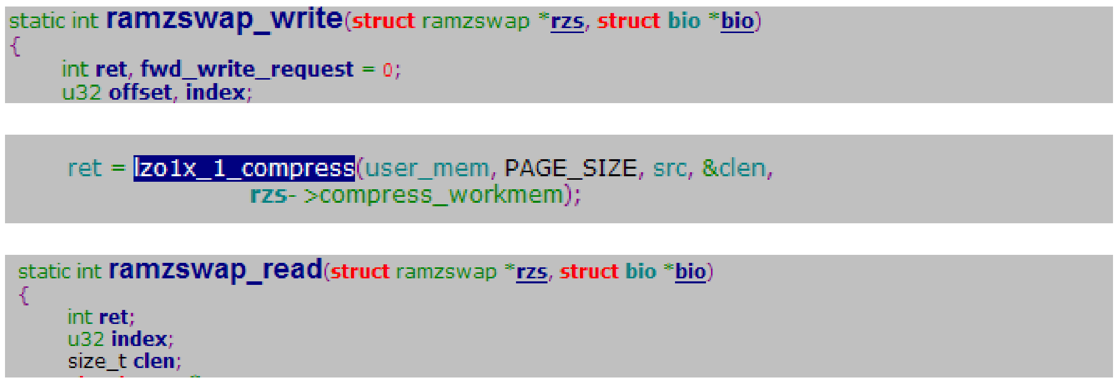
```
module_init(ramzswap_init);
```



```
static int __init ramzswap_init(void)
{
        int ret, dev_id;
        struct ramzswap *rzs;

        if (num_devices > max_num_devices) {
                pr_warning("Invalid value for num_devices: %u\n",
                                num_devices);
                ret = -EINVAL;
                goto ↓out;
        }
```

Step2 : Allocate one gendisk object (a physical block device data structure), and define the block device operations, add the IO control command.

Kernel/drivers/staging/ramzswap [12]:



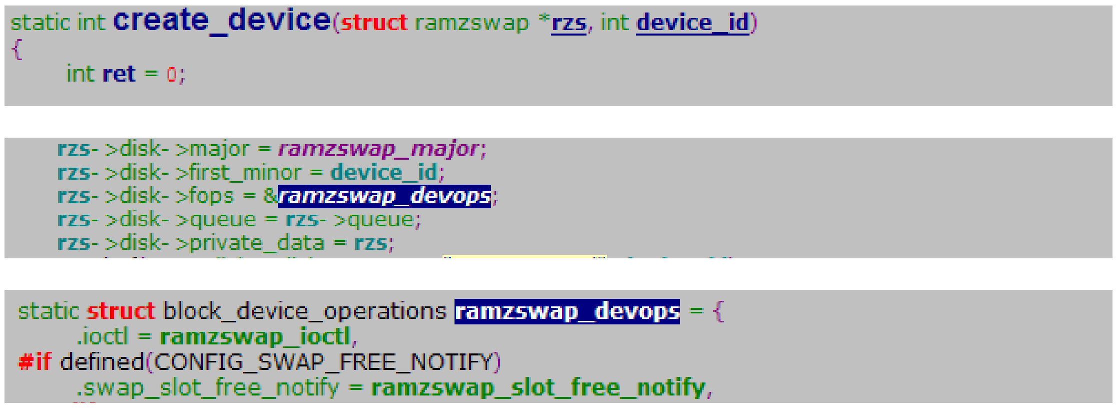
```
static int ramzswap_write(struct ramzswap *rzs, struct bio *bio)
{
        int ret, fwd_write_request = 0;
        u32 offset, index;
```

```
        ret = lzo1x_1_compress(user_mem, PAGE_SIZE, src, &clen,
                            rzs->compress_workmem);
```

```
static int ramzswap_read(struct ramzswap *rzs, struct bio *bio)
{
        int ret;
        u32 index;
        size_t clen;
```

Step3: Comp-cache use kernel module named LZO to compress data. LZO is a real time data compression library. This exhibit real time compression and decompression and it is a portable lossless data compression.

Kernel/drivers/staging/ramzswap [12]:



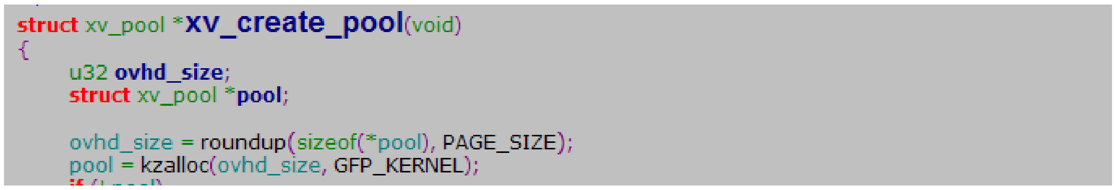
```
static int create_device(struct ramzswap *rzs, int device_id)
{
        int ret = 0;
```

```
        rzs->disk->major = ramzswap_major;
        rzs->disk->first_minor = device_id;
        rzs->disk->fops = &ramzswap_devops;
        rzs->disk->queue = rzs->queue;
        rzs->disk->private_data = rzs;
```

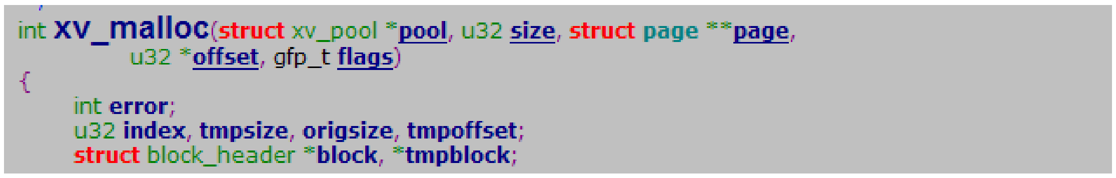
```
static struct block_device_operations ramzswap_devops = {
        .ioctl = ramzswap_ioctl,
#if defined(CONFIG_SWAP_FREE_NOTIFY)
        .swap_slot_free_notify = ramzswap_slot_free_notify,
```

Step 4: When memory manager swap out pages to swap device, Xvmalloc will allocate memory for it. The memory needed for compressed pages is not pre-allocated; it shrinks and grows on demand. On initialization, zswap creates an Xvmalloc memory pool. If the memory pool does not have enough memory to satisfy an allocation request, it grows by allocating single (0-order) pages from kernel page allocator. On freeing an object, Xvmalloc merges it with adjacent free blocks in the same page. If the resulting free block size is equal to PAGE_SIZE, i.e. the page no longer contains any object; page is released back to the kernel.

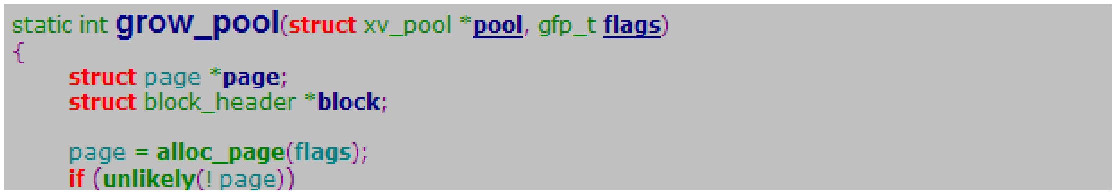
Kernel/drivers/staging/ramzswap [12]:



```
struct xv_pool *xv_create_pool(void)
{
	u32 ovhd_size;
	struct xv_pool *pool;

	ovhd_size = roundup(sizeof(*pool), PAGE_SIZE);
	pool = kzalloc(ovhd_size, GFP_KERNEL);
```



```
int xv_malloc(struct xv_pool *pool, u32 size, struct page **page,
		u32 *offset, gfp_t flags)
{
	int error;
	u32 index, tmpsize, origsize, tmpoffset;
	struct block_header *block, *tmpblock;
```



```
static int grow_pool(struct xv_pool *pool, gfp_t flags)
{
	struct page *page;
	struct block_header *block;

	page = alloc_page(flags);
	if (unlikely(! page))
```

2.5 KSM (Kernel Same Pages)

It is basically conceived and developed for the Kernel-based Virtual Machine [KVM], it was first developed for the virtualized environments. But its usage is also found good for the embedded Linux systems. KSM is implemented in the Linux as a kernel thread in the kernel (called ksmd), actually a daemon, whose job is to perform page scans in regular interval to mark duplicate pages and it merges duplicates pages to single page to free pages for other uses. It is done in a way so that user is transparent from this activity[9]. In case one of the users of the page changes the content of the page for multiple or single reason, user will receive unique copy (same as in case of Copy on Write fashion). KSM depends on higher-level applications to provide instruction on for memory regions which are good candidates for merging. KSM simply scan all anonymous pages in the system, but it is will waste of CPU and memory. So efficient way is that applications should register only those virtual areas that potentially contain duplicate pages. When KSM is enabled, it finds identical pages, keep one page in a write-protected "Copy On Write" fashion and then free for other uses. In the KSM, pages are managed using the technique of red-black trees, one is used as temporary. The first rb tree, is unstable tree, it is used to store new pages that are not yet stable. In other words, pages that are candidates for merging (unchanged for some period of time) are stored in the unstable tree. Pages kept in the unstable rb tree are not write protected. The second one, ie the stable rb tree, it stores those pages that have been found to be stable and merged by KSM.

To find if a page is non-volatile or volatile, KSM uses a 32-bit checksum. After scanning a page, its checksum is calculated and stored along the page. On immediate next scan, if recently computed checksum not matching with previously generated checksum, the page contents is changing ie is is volatile and it is not a relevant for merging.

2.5.1 Work Flow of KSM

The KSM algorithm is based on rbtrees, one is stable tree and other one unstable tree. Using two trees is an enhancement and it increases the chance of instantly sharing the pages that are good candidates for sharing as well as reduce the instability in the unstable tree.

Kernel thread scans every anonymous page scanned, and starts searching exact match in the stable tree which contains information of shared pages. If a match is found in the stable tree, the anonymous page is merged with the KSM page found in the stable tree. If no match is found in the stable tree, KSM checks content is changed recently comparing the checksum[9].

Bad case if the checksum changed since the last KSM succeeded, KSM updates the checksum and will defer the search of the unstable tree to the next KSM cycle (in anticipation that the checksum won't change again). This save CPU by avoiding adding or merging to the unstable tree pages whose contents are volatile ie changes quite often. If checksum remains constant KSM attempts searching unstable tree that contains anonymous pages scanned in previous cycle but not merged by KSM. If a match is found in the unstable tree KSM merges the anonymous page, with the anonymous page in the unstable tree, and the resulting KSM merged page is added to the stable tree (the anonymous page found in the unstable tree is removed from the unstable tree and freed). If match is not there in the unstable tree KSM adds the page to the unstable tree[6].

This "checksum" is nothing to do with the KSM algorithm. The "checksum" is only an heuristic to keep the unstable tree more stable and to avoid wasting CPU time with un matching candidates. Even if we don't use the checksum, the algorithm would work. If a page its content frequently, we'll likely only waste CPU. It is because a copy-on-write page fault will happen immediately loosing the benefits of sharing[6].

A bit by bit memory pages comparison is a CPU intensive task. Memory scanning frequency needs to meet the workload demand, otherwise it will lead to high CPU load. there are few parameters which if tuned properly will control the CPU load and hence the power performance. These are described in next section.

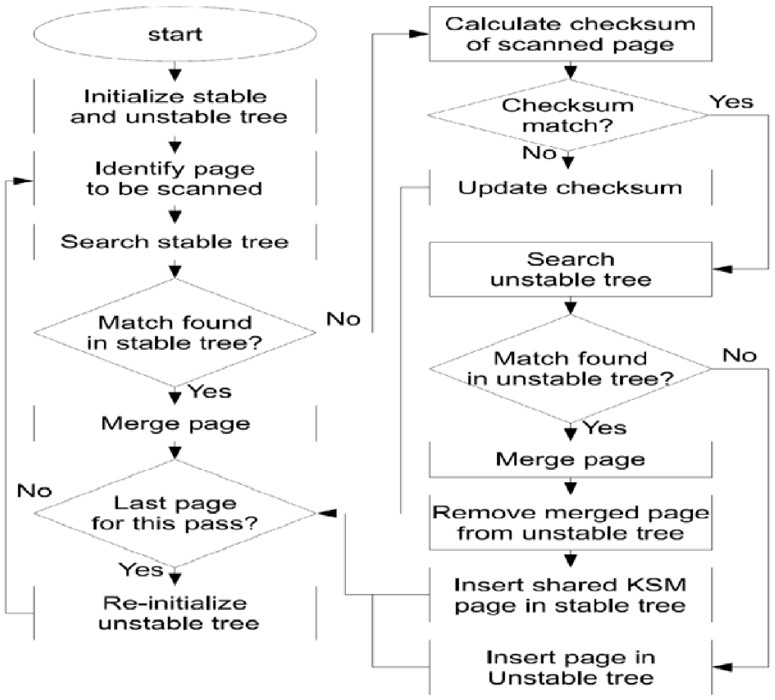

start
Initialize stable and unstable tree
Identify page to be scanned
Search stable tree
Match found in stable tree?
No
Yes
Merge page
No
Last page for this pass?
Yes
Re-initialize unstable tree
Calculate checksum of scanned page
Checksum match?
Yes
No
Update checksum
Search unstable tree
Match found in unstable tree?
No
Yes
Merge page
Remove merged page from unstable tree
Insert shared KSM page in stable tree
Insert page in Unstable tree

Figure 2-16 KSM Working Flow Chart

2.5.2 KSM Tunable Parameters

After loading the KSM module, KSM becomes operational. Relevant statistics of the KSM are located at the location /sys/kernel/mm/ksm directory. Within this directory there is separate file for each parameter specifying the current state of each parameter. The performance of KSM depends on the understanding of relation among these parameters [10].

full_scans: For memory areas for which duplications is to be indentified needs to be registered. These areas are frequently scanned. full_scans indicates the number of times memory areas has been scanned. if this number is changing and other parameter pages_shared remains same, it means we are doing useless scanning as CPU utilized in comparison but no new areas are with common pages are identified.

pages_shared: In KSM for the duplicated copies of the pages unique copy is actually given the memory. Like process A and B have five pages which are having common contents (no other process in the system has common contents)then pages_shared count is 5. This gives us count of shared pool pages. If count is N and page size is PAGE_SIZE then NxPAGE_SIZE gives us the total size that KSM is using.

pages_sharing: Number of common pages corresponding to pages_shared. This value is indicator of numbers of pages shared using KSM. A high ratio of pages_sharing vs pages_shared means KSM is working effectively. For eg zeroed pages can be shared many times but pages with encrypted data or randomized are shared few times.

pages_to_scan: As mentioned above to memory regions register them self for identification of duplicated pages. Pages in these regions are scanned periodically by ksmd. pages_to_san parametes is count of number of pages that will be scanned by KSM in each pass ie in each periodic scanning.

pages_unshared : This gives the pages which cannot be shared that is they are unique. This is indicator of that KSM effort wasted in scanning of those pages.

pages_volatile: This value indicate the number of pages that have content which is changing with high frequency. If this count is high it is indication that running process are not good candidate of memory sharing.

2.6 Low Memory Killer

LMK(Low memory killer) is a driver that android add in kernel, the mechanism is very similar to kernel's OOM killer. They will kill least important process to free memory when in low memory case.

2.6.1 OOM Killer

When after shrink cache and swap out pages, the system also has not enough memory, it is under tight memory situations, the out-of-memory killer (OOM) will be activated and selects a process to killing. User have no control over the process chosen for killing. From system prospective the process selected for killing might be the one should always be in the main memory. The process killed may a system process. To prevent highly needed process from killing, a greater degree of control is required over OOM.
The application developer always wants so ways to tell the kernel the importance of the process to be selected for killing. To give batter control the /proc/<pid>/oom_adj parameter was created under proc file system to prevent important processes from being killed by the system and define a policy for the process selection to be killed. The policy defines the ranges from -17 to +15. These ranges are the possible values for the OOM control parameter oom_adj. Higher score, means most likely hood of the process to bee selected by the OOM-killer to be killed. If parameter oom_adj is set to -17, the process is never selected by kernel for OOM-killing.

oom_adj as mentioned is the tuning node just to inform the OOM-killer for the desire ability to keep in memory. "Badness Score is the ultimate parameter to decide for the selection by OOM-killer. The candidate process selected for killing in an out-of-memory situation is selected based on its badness score. The badness score is reflected in /proc/<pid>/oom_score.

This value is determined on the basis that the system loses the minimum amount of work done, recovers a large amount of memory, doesn't kill any innocent process eating tons of memory, and kills the minimum number of processes (if possible limited to one)[4]. Parameters for the badness score are original memory size of the process, CPU time (utime + stime), the run time *(uptime - start time) and oom_adj value. The longer a process live smaller the score and more memory the process uses, the higher the score.*

.

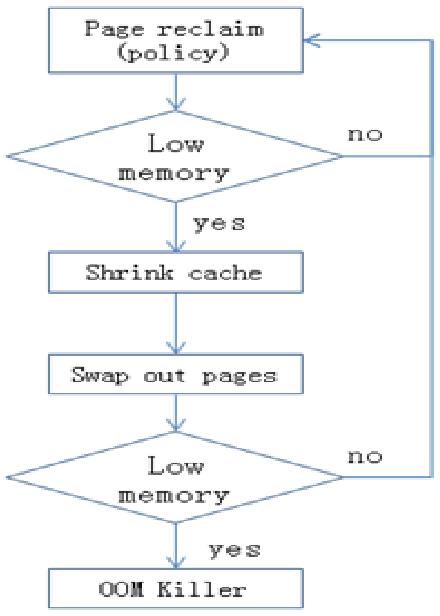


Figure 2-17 OOM Working Flow Chart

2.6.2 Low Memory Killer

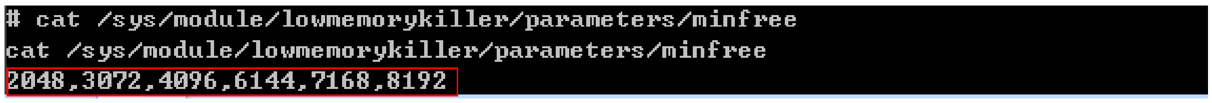
OOM-killer is not sufficient solution for devices involving high degree of multiprogramming such as Mobile Phones. On Mobile Phones we have very limited size screen and only few processes can be in for ground .Back Ground processes keep supporting the for ground processes. Algorithmic criteria of calculating the badness is not sufficient to predict the right process to be killed. Moreover Android wants the greater flexibility to be given to the user via Android Framework. This way Android proposes a new philosophy of LMK which handles the low memory situations effectively. On low memory situation LMK handles the memory pressure without invoking the OOM-killer. LMK deferred the OOM-killer participation and keep great degree of control in the hand of Framework. LMK is implemented as the kernel driver in Android. LMK defines the user adjustable threshold values of the memory pressure and categories the Android Processes as shown in the table below. When the first threshold is encountered LMK kills the Empty Applications. If memory pressure is further increased LMK checks for the other processes in order from highest to lowest. Foreground apps has the lowest weight and are last to be picked in the order.

The low memory killer will kill process according to process LMK adjustments as below, the higher the LMK value, more likely the associated process is to be killed.

Name	Weight	Note
FOREGROUND_APP	0	Show in screen or some system process
VISIBLE_APP	1	User visible but not foreground (widget, IME)
SECONDARY_SERVER	2	System service (Gmail internal memory, contactor internal memory)
HIDDEN_APP	7	Background process (browser, reader)
CONTENT_PROVIDER	14	Content provider, used for other process
EMPTY_APP	15	Empty app, without service/content providing

Figure 2-18 Android Process Priority

The thresholds of low memory is set in file : /sys/module/lowmemorykiller/parameters/minfree.



```
# cat /sys/module/lowmemorykiller/parameters/minfree
cat /sys/module/lowmemorykiller/parameters/minfree
2048,3072,4096,6144,7168,8192
```

Figure 2-19 Threshold setting in Android

The 6 value is memory thresholds for 6 types applications, take empty app for instance, the value is 8192, so when the phone’s free memory is less than 32M(8192 pages), the LMK will kill empty app to get enough memory.

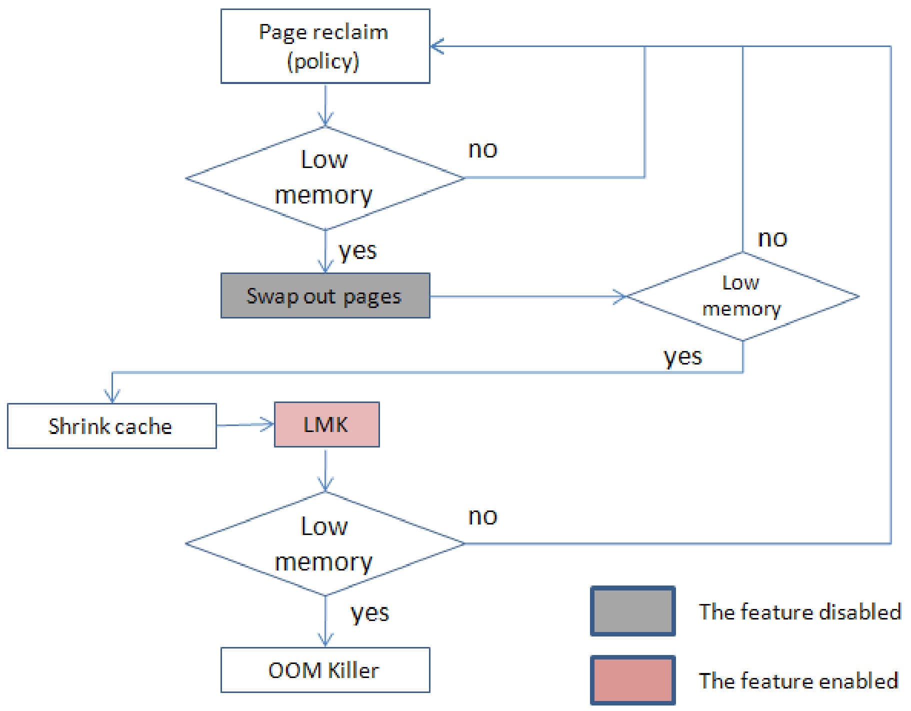
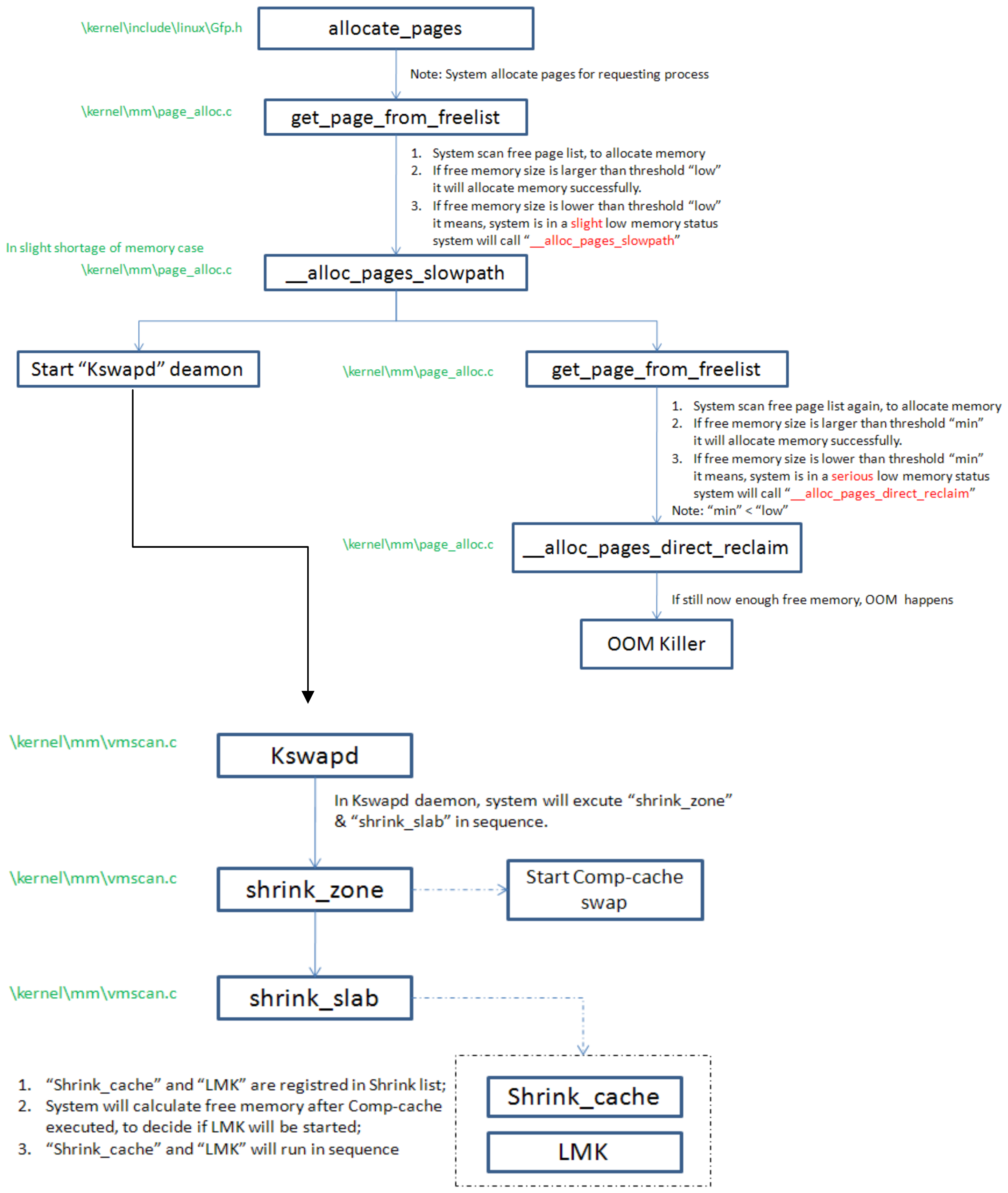


Figure 2-20 Android LMK Working Flow Chart

2.6.3 Summary of calling process in low memory status

When system finds it’s hard to allocate memory for each process requesting, system is in low memory status. Everything start from function----“allocate_pages()”.



\kernel\include\linux\Gfp.h
allocate_pages
Note: System allocate pages for requesting process
\kernel\mm\page_alloc.c
get_page_from_freelist
1. System scan free page list, to allocate memory
2. If free memory size is larger than threshold "low" it will allocate memory successfully.
3. If free memory size is lower than threshold "low" it means, system is in a slight low memory status system will call "__alloc_pages_slowpath"
In slight shortage of memory case
\kernel\mm\page_alloc.c
__alloc_pages_slowpath
Start "Kswapd" deamon
\kernel\mm\page_alloc.c
get_page_from_freelist
1. System scan free page list again, to allocate memory
2. If free memory size is larger than threshold "min" it will allocate memory successfully.
3. If free memory size is lower than threshold "min" it means, system is in a serious low memory status system will call "__alloc_pages_direct_reclaim"
Note: "min" < "low"
\kernel\mm\page_alloc.c
__alloc_pages_direct_reclaim
If still now enough free memory, OOM happens
OOM Killer
\kernel\mm\vmscan.c
Kswapd
In Kswapd daemon, system will excute "shrink_zone" & "shrink_slab" in sequence.
\kernel\mm\vmscan.c
shrink_zone
Start Comp-cache swap
\kernel\mm\vmscan.c
shrink_slab
1. "Shrink_cache" and "LMK" are registred in Shrink list;
2. System will calculate free memory after Comp-cache executed, to decide if LMK will be started;
3. "Shrink_cache" and "LMK" will run in sequence
Shrink_cache
LMK

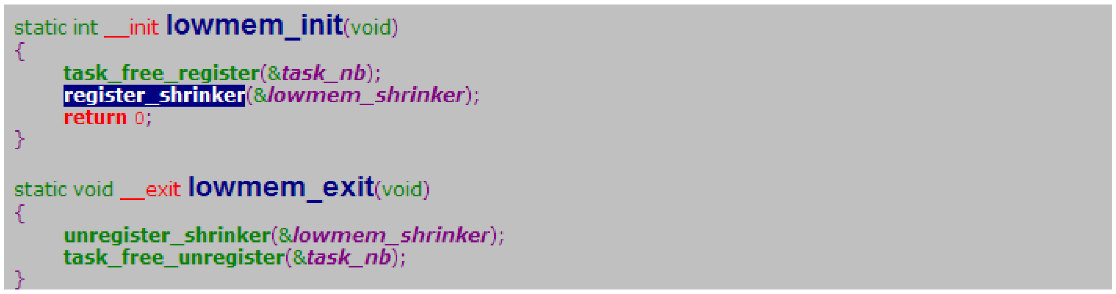
Figure 2-21 Android Low Memory Working Flowchart

2.6.4 Low Memory Killer Code

Low memory killer is a shrinker driver that register in kernel shrinker list. In Linux kernel, Swapping the pages of user space applications is not the kernel's single method of freeing memory space. Caches shrinking often results in batter gains.

Step 1:Shrinker methods are the functions in the kernel and these are registered dynamically. When memory is under pressure the kernel invokes the shrinker registered functions to free the memory.Low memory killer register in the kernel shrinker list when the module initialize.

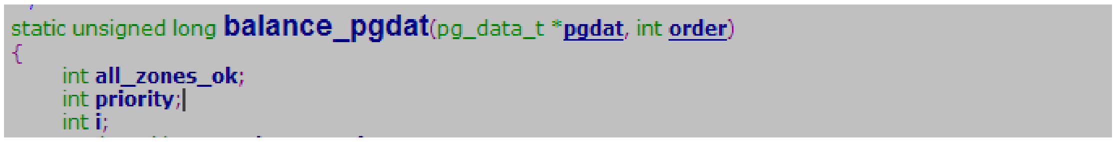
Kernel/drivers/staging/android/lowmemorykiller.c [12]:



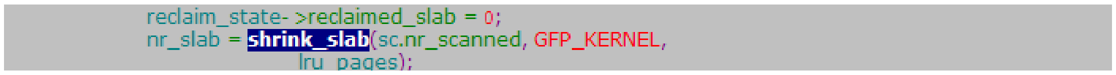
```
static int __init lowmem_init(void)
{
        task_free_register(&task_nb);
        register_shrinker(&lowmem_shrinker);
        return 0;
}

static void __exit lowmem_exit(void)
{
        unregister_shrinker(&lowmem_shrinker);
        task_free_unregister(&task_nb);
}
```

Step 2:A daemon thread (Kswapd) runs in the background, it will do nothing when internal free memory is enough. When the system is in shortage of memory situation, this daemon thread (Kswapd) will be wake up; after that, Kswapd thread will call balance_pgdat() to reclaim pages. (swap out pages and shrink cache…)

Kernel/mm/vmscan.c [12]:



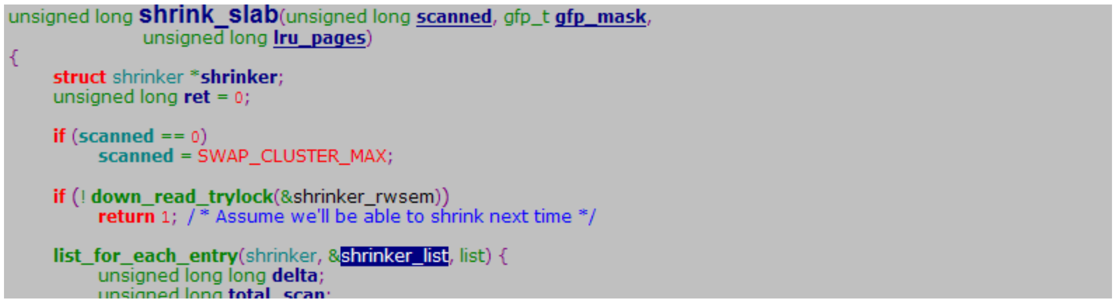
```
static unsigned long balance_pgdat(pg_data_t *pgdat, int order)
{
        int all_zones_ok;
        int priority;
        int i;
```



```
                reclaim_state- >reclaimed_slab = 0;
                nr_slab = shrink_slab(sc.nr_scanned, GFP_KERNEL,
                                lru_pages);
```

Step 3:The function of shrink_slab() will call each shrink functions in shrinker list to shrink cache.

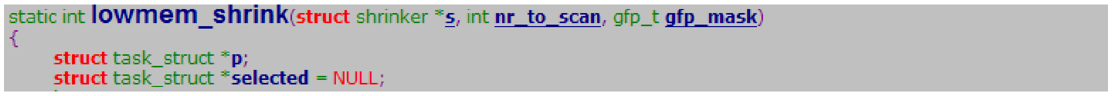
Kernel/mm/vmscan.c [12]:



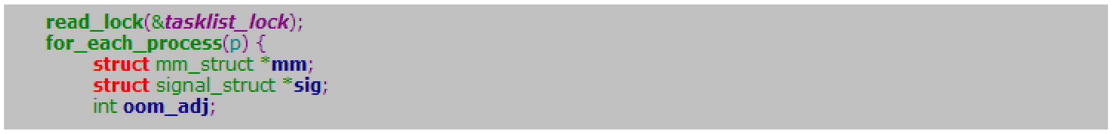
```
unsigned long shrink_slab(unsigned long scanned, gfp_t gfp_mask,
                        unsigned long lru_pages)
{
        struct shrinker *shrinker;
        unsigned long ret = 0;

        if (scanned == 0)
                scanned = SWAP_CLUSTER_MAX;

        if (! down_read_trylock(&shrinker_rwsem))
                return 1;  /* Assume we'll be able to shrink next time */

        list_for_each_entry(shrinker, &shrinker_list, list) {
                unsigned long long delta;
                unsigned long total_scan;
```

Step 4:Then the low memory killer shrinker will be called. It will kill less important application according to OOM value and memory usage.

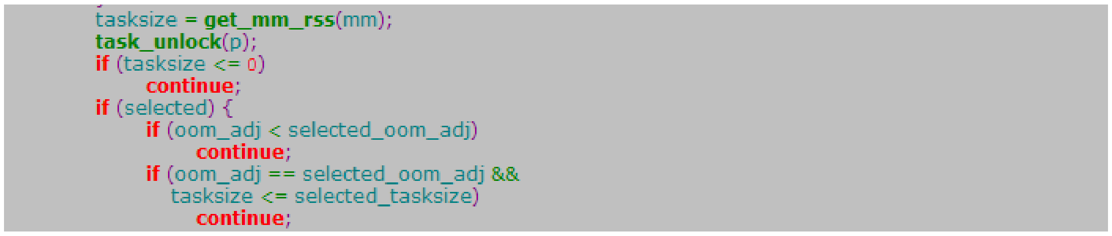
Kernel/drivers/staging/android/lowmemorykiller.c [12]:



```
static int lowmem_shrink(struct shrinker *s, int nr_to_scan, gfp_t gfp_mask)
{
        struct task_struct *p;
        struct task_struct *selected = NULL;
```



```
        read_lock(&tasklist_lock);
        for_each_process(p) {
                struct mm_struct *mm;
                struct signal_struct *sig;
                int oom_adj;
```



```
                tasksize = get_mm_rss(mm);
                task_unlock(p);
                if (tasksize <= 0)
                        continue;
                if (selected) {
                        if (oom_adj < selected_oom_adj)
                                continue;
                        if (oom_adj == selected_oom_adj &&
                            tasksize <= selected_tasksize)
                                continue;
```

Chapter 3

New Policy for cope up Low Memory Situation in Android Operating System

In the previous chapters we developed the understanding of the Linux and Android Memory Management, how Android utilizes the LMK in case of memory pressure. But frequency of calling LMK itself is an overhead. As it removes the background applications from the memory and if second time application is re-entered it takes longer time causing sluggish behavior. So here we are defining the new LMK algorithm which ensures that frequency of LMK is reduced and always sufficient RAM is available. The new LMK Policy takes the benefits of the Com Cache and KSM techniques. These techniques was originally proposed for different purposes, like KSM was proposed for memory saving in the Virtualized environment and Com Cache was proposed for reducing the OMM killer optimization. The New LMK policy improvements are done at the three levels , in first level KSM major drawback of the CPU utilization is reduced by selection of the processes which have good probability of the same pages. KSM is modified to work well with Android. In second level Com Cache pages with compression ration more than 0.5 is selected and in final level LMK new policy is proposed with incorporates the advantages of the in kernel memory compression techniques.

3.1 Drawback of Comp-cache & LMK & KSM

When we use comp-cache to extend android phone physical memory, the swap feature need to be enabled in kernel. The swap will work with comp-cache, if the android system is in acute memory shortage, the kernel will swap out pages to comp-cache.But Linux kernel swap in-active pages from LRU list, regardless which process does this page belong to. Some in-active pages are swapped in, but later, they will be used again and swap out; It will cost compress/decompress, swap in/out time.

Following table shows the time cost for one page compress/de-compress:

Function	Time
Compress	49us
De-compress	27us

Figure 3-1 Time Cost of Compress/De-compress One Page

Comp-cache compress & swap in-active pages from LRU list:

In Android, these pages can be from an “empty process” or an “active process”;

- Comp-cache just picks the most in-active pages from LRU list;
- Comp-cache doesn’t know which process these pages belong to.

Consider that one in-active page (A) is belong to an active process:

- “Page A” is picked by Comp-cache, and be swapped;
- After a while, “Page A” is needed by the process to be active again;
- “Page A” will be firstly decompressed, then, swap out;
- Extra decompress, swap out time are needed to access “Page A”.

To improve the user experience, Android keeps closed application as “Empty process”, (whereas, original Linux kernel, just kill the corresponding process, and free the physical memory to system), it makes memory easily using out. More and more applications are running & closed, more and more memory is used out.

When Android system is in low memory status, LMK will be invoked to kill low priority process, free the memory to system:

- “Empty process” will be killed firstly, leading to:
  - ✓ Android cannot start corresponding “closed application” quickly at next time;
  - ✓ It’s meaningless for Android to keep “empty process” for “closed application”.

- Then, “active process” will be killed according to priority:
  - ✓ Some applications will be closed unexpected for user;
  - ✓ Take a bad user experience.

When KSM is enabled in the Android System it performs following in the idle thread

- Initializes stable and unstable tree
- Identify all the processes in the systm
- Identify page to be scanned
- Search it in the Stable Tree
- if match is not found in the stable tree , its check sum is calculated
- checksum calculated does match with the previous checksum search in unstable tree
- if found in unstable tree merge pages and moves to stable tree
- checksum calculated does not match insert it in in unstable tree
- Viewing the Figure < 2-21> is good to understand the above steps

The above steps execution is not free it consumes lot of CPU, and drains the battery not acceptable for the Mobile Phones, the out of RAM saves is much smaller then CPU usage.

3.2 LMK and Com-Cache Optimization

Android LMK as originally introduce to combat the memory pressure. In its present form it good but this can be further improved with the ideas as described in the figure below. As shown in the figure a module describing "New Policy" is introduced. This policy is shown as separate module in the figure below. Implementation wise LMK and New Policy module are integrated one. The working of the policy is described in the form of the steps:

Step1: When the memoty processor is increased in the system first of all system shrinks the caches. Android maintain the caches as the pre allocated structures of the frequently used objexts such as inodes, process control block, thread, Virtual memory areas. The pre allocation is done to fast the execution.

Step 2 : If system is not able to recover form the memory pressure after shrinking the caches LMK is called.

Step 3 : LMK Algorithm is executed based on the new policy, The new policy integrate the KSM and Com Cache techniques to counter the memory pressure first then decides for killing the

Step 4 : Comcache compresses the pages using LZO algorithms. It takes the 4KB pages contents and apply LZO to compress them to save the meory

Step 5 :Checks the Physical pages of the procees in RAM and compress them using the comcache. If comression is more than 0.5 swap them to the "Ramswap" the block of memory resrved to store the compressed pages.The details algorithms for identification of pages of the processes are described separtely below.

Step 6: Selected pages are stored in the "Ramzswap"

Step 7: Even after the compression of pages if memory pressure is not reduced the the normal LMK algorithm of killing the processes is executed , with order of killing started form backgrount processes, providers processes , hidden app, secondary server and finally foreground processes.

Step 8: Calling OOM is the last option, LMK is introduced in the Android to avoid the OOM execution. This is used as the last option to counter memory pressure. When modified LMK failed to reduce the memory pressure OOM is called.

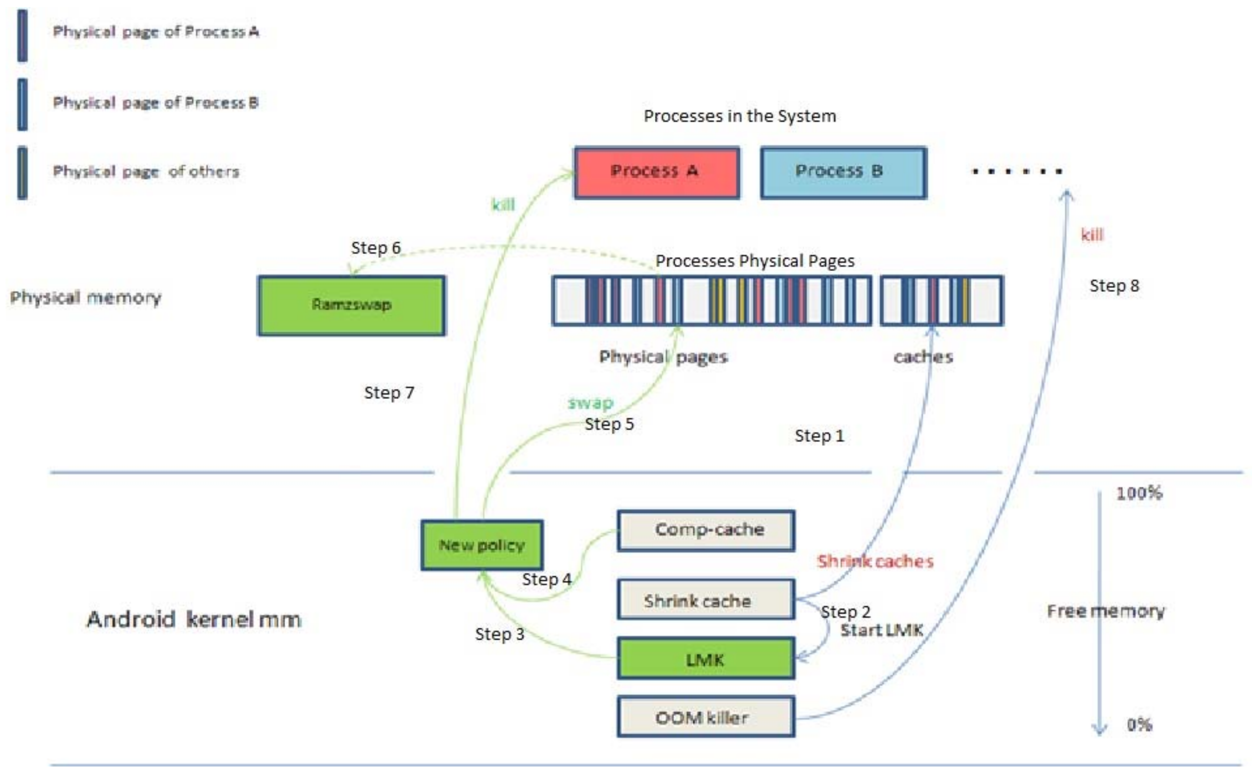


Figure 3-2 New LMK Policy

– Virtually enlarge internal memory by compress & swap in-active physical pages;

– Reduce the time cost effect caused by compress & swap;

– Keep empty process in memory as long as possible (keep the benefit of empty process);

– Always firstly compress & swap all physical pages of empty process;

✓ After compression, empty processes only occupy little memory area;

✓ Meanwhile, keep empty process in memory can restart corresponding application more quickly;

✓ An empty process will only be killed at the very last second;

✓ Physical pages that belong to an empty process will not be re-active again, it can reduce the decompress/swap time for a page;

– New LMK & Comp-cache policy;

✓ Compress & swap empty process when system is in idle state; As soon as an empty process exists, system will compress & swap it;

✓ Avoiding the situation that: compress/swap take extra time/CPU working cost

- ✓ When LMK begin to kill process, keep all memory is occupied by active process; which improve utilization of memory resource.

LMK Modified Algorithm is as followed:

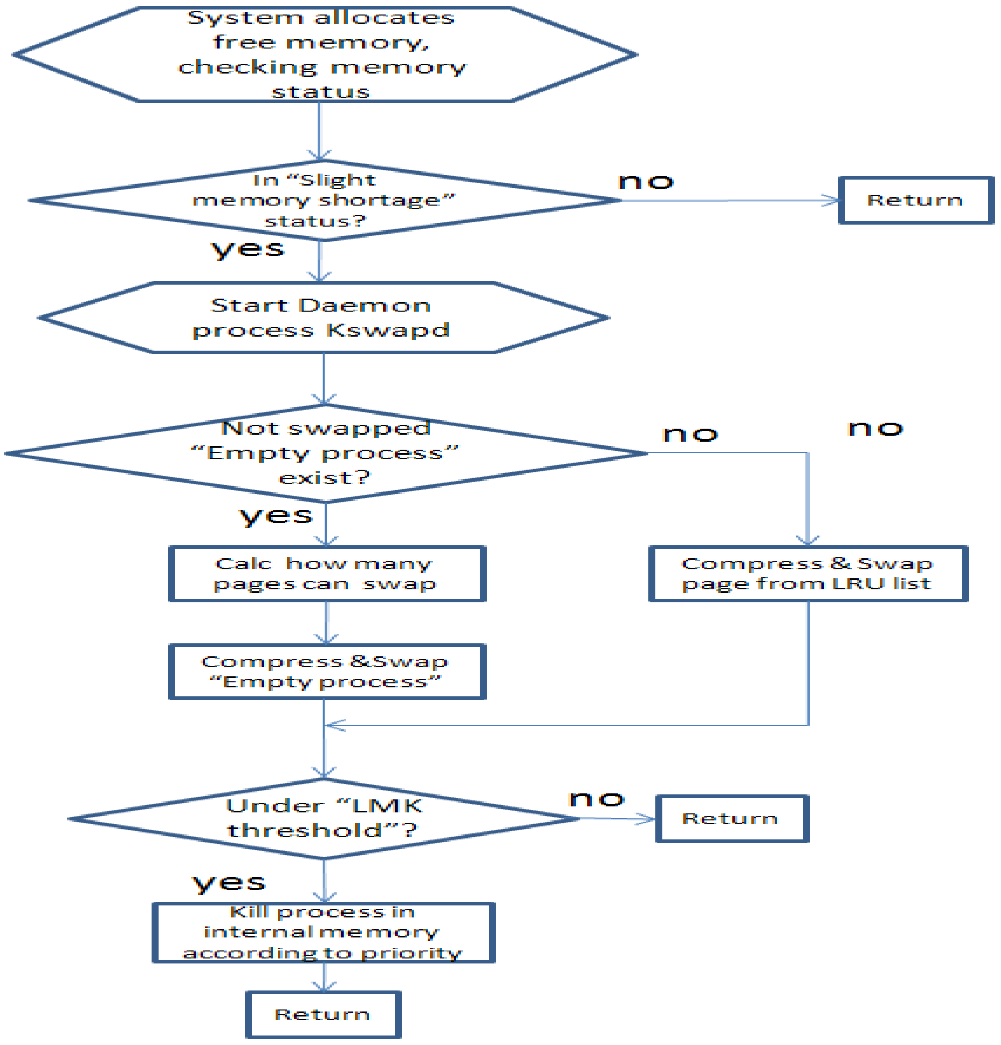

System allocates free memory, checking memory status
In "Slight memory shortage" status?
no
Return
yes
Start Daemon process Kswapd
Not swapped "Empty process" exist?
no
no
yes
Calc how many pages can swap
Compress & Swap page from LRU list
Compress &Swap "Empty process"
Under "LMK threshold"?
no
Return
yes
Kill process in internal memory according to priority
Return

Figure 3-3 New LMK Policy Flow Chart

The optimization of comp-cache and swap is to compress & swap physical pages based on process level. The difference is shown in **Figure <3-4>**

Solution	Benefit	Shortage
Original	Enlarge internal memory virtually	Cost compress/decompress, swap in/out time
Optimized	Enlarge internal memory virtually Swap empty app firstly, reduce the time cost by swap in/out Keep Android & Comp-cache's benefit at same time	The code of swap will be more complex

Figure 3-4 Comparison between Original Comp-cache & Optimized Comp-cache

New mechanism that can swap all physical pages that belong to a dedicated process should follow following steps:

- Search "Empty process";
- Search all physical pages belong to a dedicated "Empty process";
- Search the shared pages in those physical pages;
- Define new swap page list;
- Update LRU list.

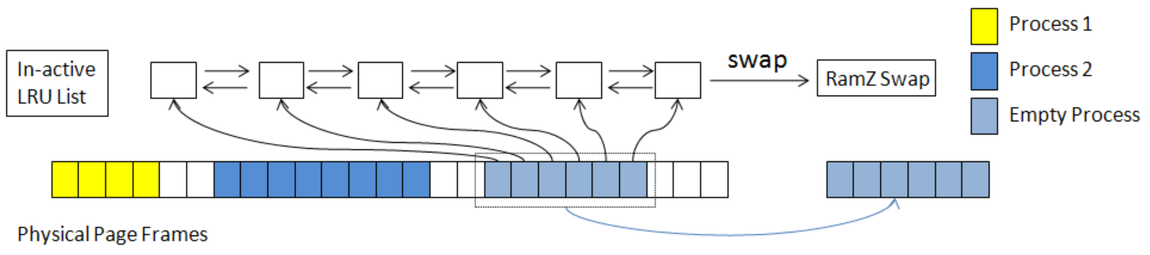


Figure 3-5 Compress & Swap for Dedicated Process

Figure <3-5> show the work flow of swap pages to comp-cache based on android application priority. Firstly, create a empty application table, it stores all of the empty applications. When the system is in idle status, invokes the search mechanism to find out the pages of empty application, then unmap the pages, finally update the LRU list and swap pages to disk.

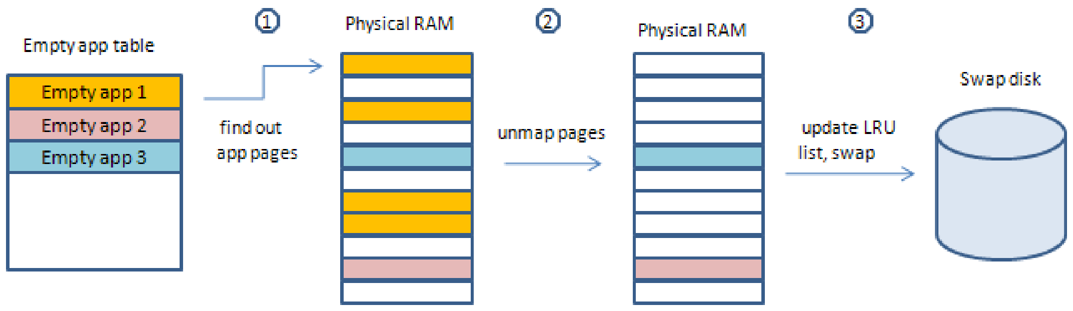


Figure 3-6 Sequence to Search Physical Pages I

1st step: to search all existing "Empty process" in Android system:

- When Android application exits, it will change the oom_adj value to 15, but not free the memory;
- Android has a process record list to store all Android applications information, scan this list, can search out all available Empty process;
- The sample code is like below, check the process's oomAdj, if it is equal or greater than EMPTY_APP_ADJ (15), it should be empty process.

2nd step: Search all physical pages belong to a dedicated process:

We will use the process structure to find the page table, then find the application pages that in physical memory.

- Find out the mm_struct with the empty process's task_struct;
- Use the mm_struct to find the process's virtual address (vm_area_struct);
- The process's virtual address can get the (PGD) Page Global Directory
- Use the PGD and virtual address can find (PUD) Page Upper Directory
- Then find the Page Middle Directory (PMD) by using PUD and virtual address;
- Find the Page Table (PTE) by using PMD, and get the physical page finally.

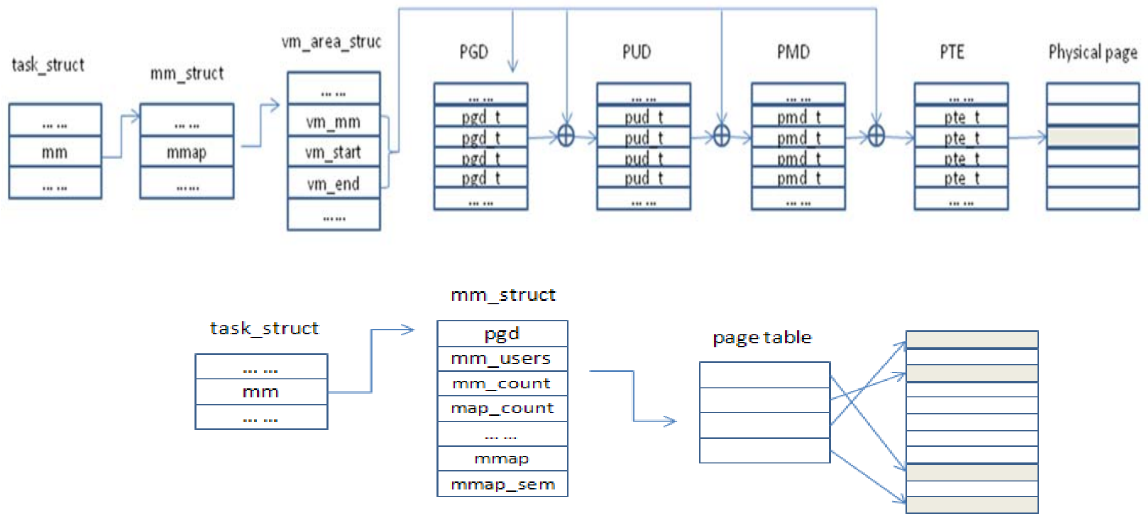


Figure 1-7 Sequence to Search Physical Pages II

3rd step: Find all referenced logical pages of a dedicated physical page:

We will use the kernel function of try_to_unmap(), it will check all the page's mapping tables, if the page is not locked, the page can swap out. Before swap , the system will release the page mapping connection from all process' s page table.

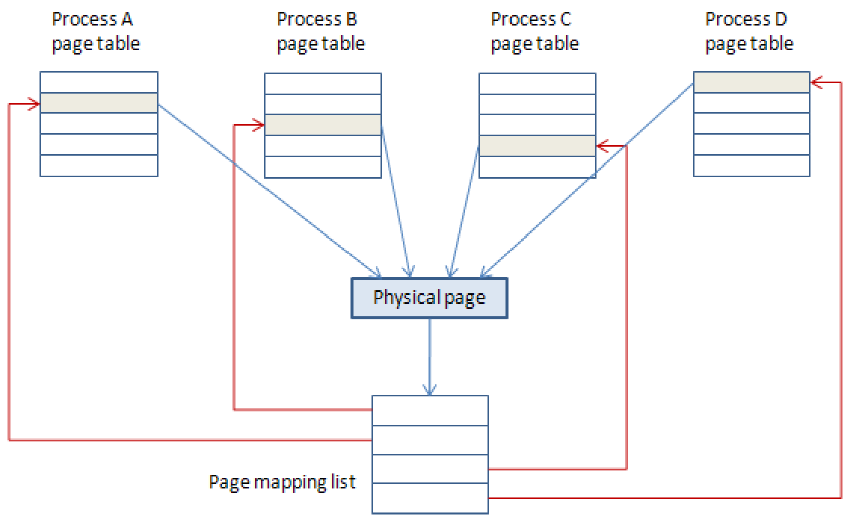


Figure 3-8 Reserve Mapping

4th step: Setup swap page list, update in-active LRU list:

We can set the PG_active and PG_reference to 0, then call the kernel function of lru_cache_add() to update LRU list, finally invoke swap mechanism.

- If the page is in LRU list and can be moveable, delete the page from the list;
- Add the page into new temporary list;
- Enable the swap mechanism, swap the pages in new temporary list to swap disk;
- If the pages swap to disk failed, delete from the temporary list, then give back to LRU list.

As a result, we test the ratio that how many physical pages in a process can be compress & swap to Ramzswap. As each process can consist different shared library, shared data, these pages cannot be compress & swap, so the ratio of compressed pages of a process is different to each other

Application	Total pages	Compress & Swap pages	Swap Ratio	Compress Ratio
Wow fish	6687	3169	47.4%	41.4%
Angry birds	11283	6323	56%	56.9%
Plants Vs Zombies	24023	7955	33.1%	52%
Trial X	11991	8433	70.3%	51.8%
Cut the rope	18574	11799	63.5%	28.4%
Angry bots	23587	20401	86.5%	55.9%
Total	96145	58080	60.4%	47.73%

Figure 3-9 Swap Ratio for Dedicated Application

Note:

- Swap Ratio: (Compress & Swap pages) / (Total pages)
- Compress Ratio: the average compress ratio for one page

3.3 KSM Optimization

As software for mobile phones becomes more complex the amount of needed random memory (ram) increases, too. While enlarging ram-size of common desktop computers is unproblematic and cheap, it is difficult for mobile device as it effects two of their important attributes: power consumption and size. Ram is a constant power consumer; even when a mobile phone is in standby mode its ram must be powered. For small amounts of ram its power consumption is low compared to other components. Nevertheless increasing RAM size raises the power consumption of mobile devices and increases their size, in case of using a larger battery. As the single parts of mobile devices are packed together extremely narrowly, adding a single (memory) chip might imply a larger housing in any case. Multitasking operating systems are prone to load the same pieces of data into multiple physical pages (page duplication) as lots of programs, accessing partially equal data, run in parallel. Reducing page duplication yields a reduction of memory consumption, as all but one page containing the duplicated page content can be freed. In this theses we investigate the KSM on Android mobile phones and analyze the memory saving potential.

As mentioned above when discussed the drawbacks of the KSM. For Android if we exploit the Android Process Management, we can find something interesting. For KSM algorithm to work effectively it is important that we reduce the number of scans and get maximum duplicated pages. This is only possible if scanning is performed on the process whose pages are :

- Less Volatile means does not change in the subsequent scanning
- Processes which have high tendency to be similar

With the exploitation of the above characteristics we are proposing a scheme that decreases the CPU cost incurred in computation. All these experiments are performed on the Galaxy Nexsus with proper real Workload.

Optimization #1

As mentioned in Chapter 2 , the Android Framework provides one foreground application and many background applications. Since phone/tablet is maximum 10" not the complete TV , so only one application is active in foreground rest of applications are not inert but in background. One important consideration form the memory point of view is that background applications are changing their contents from the memory point of view it is only the foreground application which is updating their memory content frequently. So this helps in identifying the volatile pages process.

If KSM only targets the background processes, the process pages only need to be scanned once as till the time process is in background its memory contents are not going to be changed. So during subsequent round the page is not scanned.

The results are shown in Chapter 4 , it shows 98% CPU saving with the same amount of memory saved without adopting the Optimization #1

Optimization # 2

Android Zygote Model : All the processes in the Android are derived from the Zygote. The concept is introduced for the fast creation of process.

Zygote is daemon process whose only task is to launch applications. So Zygote is the parent of all App process. When app_process launches Zygote, it first creates DVM and then it calls Zygote’s main () method. On starting the Zygote it loads all required Java classes and other resources, it starts "System Server" and opens */dev/socket/zygote* socket which listen for requests for other starting applications. System Server is a complete detached process from its parent. Once it is created it keeps on initializing all various System Services and it starts the Activity Manager . This is how it works, this information was important to understand how KSM can effectively use it.

From /dev/socket/zygote , "zygote receive a request to launch an application. On receiving request **fork()** is called. Here lies the important stuff. When a process forks, a clone of its is created. It means replicating itself in some another memory space. This is done in a special way. Zygote, first creates an exactly same new DVM, preloaded all necessary resources and classes required Application. This really makes the process of creating a VM and load resources pretty efficient. Android uses the modified Linux kernel. The Linux Kernel implements **Copy On Write** (COW)strategy.This means is that during the fork process, no memory is actually copy to another space. It is shared and marked as **copy-on-write**. So all the libraries at the same virual addresses are same. *So if KSM know these then these pages are guaranteed candidate od page sharing*. [5]

3.3 Optimized low memory policy

In Android low memory solution, optimized Comp-cache will be invoked in two situation.

- When system is in idle status, system will scan for empty process, compress & swap them (optimized Comp-cache);
- When system is in low memory status, optimized Comp-cache & LMK will work.

System Idle status: When screen turns off (or press power key), optimized Comp-cache will start. It will firstly search empty process in system, then, compress & swap these empty process. In idle status, system will always compress & swap empty process, until the swap area is full. The working flowchart is described as following diagram:

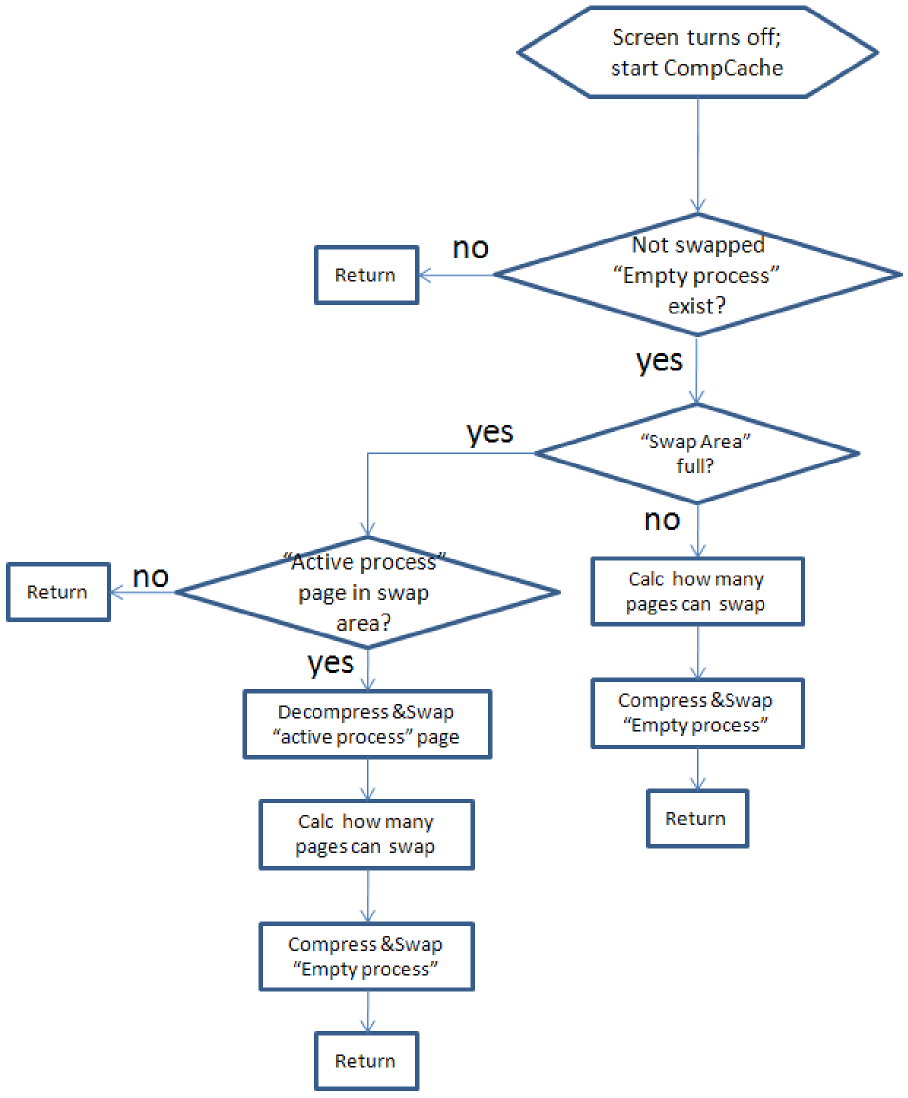

Screen turns off; start CompCache
Not swapped "Empty process" exist?
no
Return
yes
"Swap Area" full?
yes
no
"Active process" page in swap area?
no
Return
yes
Decompress &Swap "active process" page
Calc how many pages can swap
Compress &Swap "Empty process"
Return
Calc how many pages can swap
Compress &Swap "Empty process"
Return

Figure 3- 10 Optimized Comp-cache Working Flowchart in Idle Status

System low memory status: When system memory is in "slight memory shortage" status, the optimized Comp-cache will start (combined with LMK). It will firstly search the empty process in system, then, compress & swap 32 pages of the process every time. If the free memory is still low, system will start LMK, to kill process according to process priority and memory cost. Detail working flowchart is described as following:

As software for mobile phones becomes more complex the amount of needed random memory (ram) increases, too. While enlarging ram-size of common desktop computers is unproblematic and cheap, it is difficult for mobile device as it effects two of their important attributes: power consumption and size. Ram is a constant power consumer; even when a mobile phone is in standby mode its ram must be powered. For small amounts of ram its power consumption is low compared to other components. Nevertheless increasing RAM size raises the power consumption of mobile devices and increases their size, in case of using a larger battery. As the single parts of mobile devices are packed together extremely narrowly, adding a single (memory) chip might imply a larger housing in any case. Multitasking operating systems are prone to load the same pieces of data into multiple physical pages (page duplication) as lots of programs, accessing partially equal data, run in parallel. Reducing page duplication yields a reduction of memory consumption, as all but one page containing the duplicated page content can be freed. In this theses we investigate the KSM on Android mobile phones and analyze the memory saving potential.

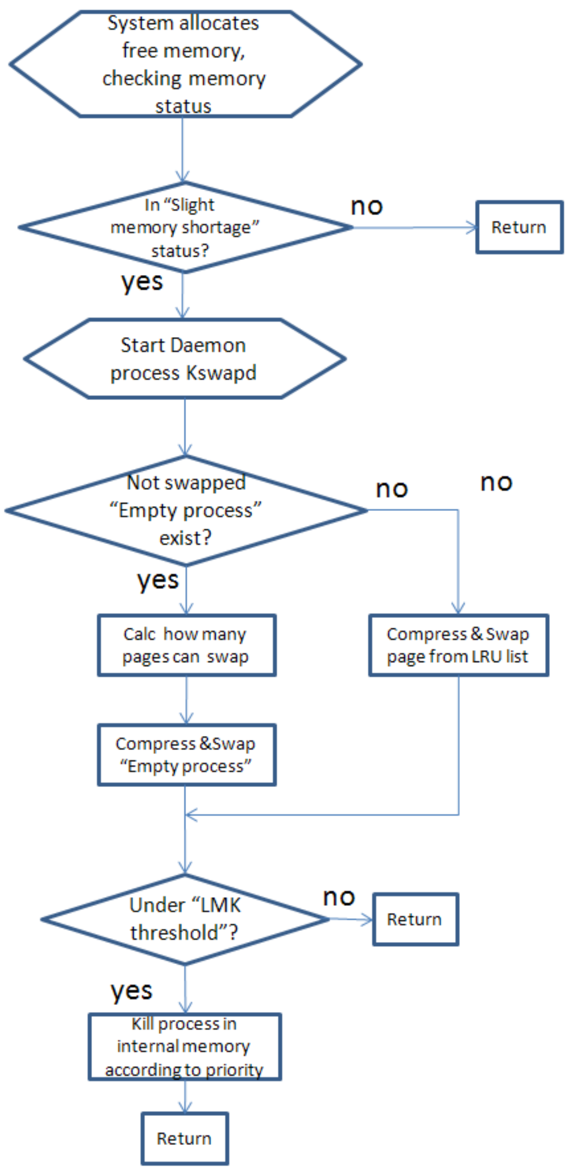

System allocates free memory, checking memory status
In "Slight memory shortage" status?
no
Return
yes
Start Daemon process Kswapd
Not swapped "Empty process" exist?
no
no
yes
Calc how many pages can swap
Compress & Swap page from LRU list
Compress &Swap "Empty process"
Under "LMK threshold"?
no
Return
yes
Kill process in internal memory according to priority
Return

Figure 3-11 Optimized Comp-cache Working Flowchart in Low Memory Status

For KSM there is not specialized change in the calling algorithm. Only change is the input to the ksmd daemon. When it scan the process Virtual address ranges as described in Optimization #2 and selection of processes as described in Optimization #1 . So selection of input to be processed giving the good results.

Chapter 4

Results and Analysis

After developing the new LMK solution for the Android System, testing the stability of the system with new algorithm is also an important task. To test the stability of the system with the new solution we are using the standard benchmark test cases. With this new algorithm we are keen to judge the stability and performance. Android provide the monkey test suite to judge the stability of the system. This test randomly launch the application in the mobile phone, start the activities , kill activities, fires intents and do number of operations on each applications to make sure system is stable. For performance tests we used the IOZONE that is memory read write test,

The final performance evaluation will cover to main area:

- The system stability test;
- The system performance test

4.1 System stability test for Com Cache

The system stability test includes two part:

- Linux IOZONE test
- Android Monkey test

For IOZONE test, the benchmark result is as following:

HW Platform: G-I9103;
SW Platform: Original Android + Optimized Comp-cache;
Block size = 4KB File size = 300MB

Test target	Write (Kbytes/s)	Re-write (Kbytes/s)	Read(Kbytes/s)	Re-read (Kbytes/s)
optimized	15974	13899	428161	518079
un-optimized	15999	14527	447303	533059

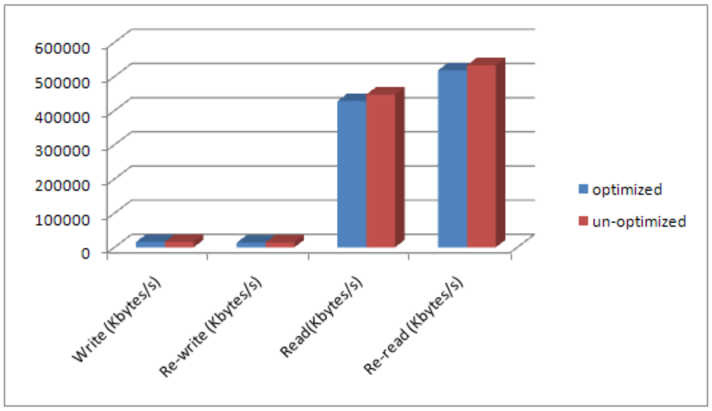


Figure 4-1 Block Read/Write (4K)

Block size = 8KB File size = 300MB

Test target	Write (Kbytes/s)	Re-write (Kbytes/s)	Read(Kbytes/s)	Re-read (Kbytes/s)
optimized	15983	14443	466141	568768
un-optimized	15721	13983	512293	585576

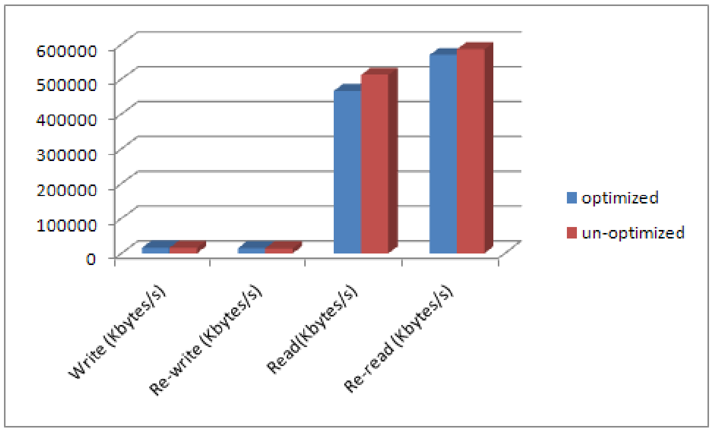


Figure 4-2 Block Read/Write (8K)

Block size = 16KB File size = 300MB

Test target	Write (Kbytes/s)	Re-write (Kbytes/s)	Read(Kbytes/s)	Re-read (Kbytes/s)
optimized	16413	14766	508073	619928
un-optimized	16854	14491	541209	619606

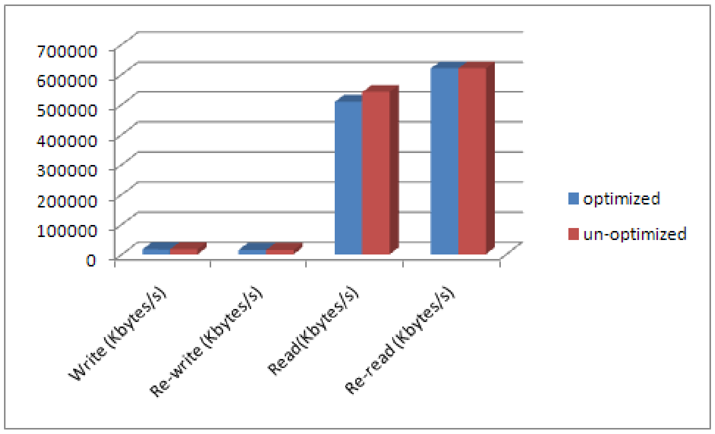


Figure 4-3 Block Read/Write (16K)

Block size = 32KB File size = 300MB

Test target	Write (Kbytes/s)	Re-write (Kbytes/s)	Read(Kbytes/s)	Re-read (Kbytes/s)
optimized	14481	15603	537463	624593
un-optimized	14395	14057	553960	634622

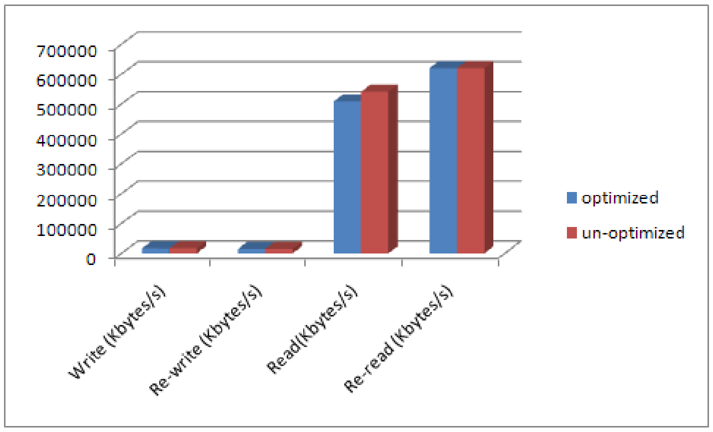


Figure 4-4 Block Read/Write (32K)

From the IOZONE test result, optimized Comp-cache does not bring side effect to system overall stability.

For Monkey test, the benchmark test result is as following:

HW Platform: G-I9103;

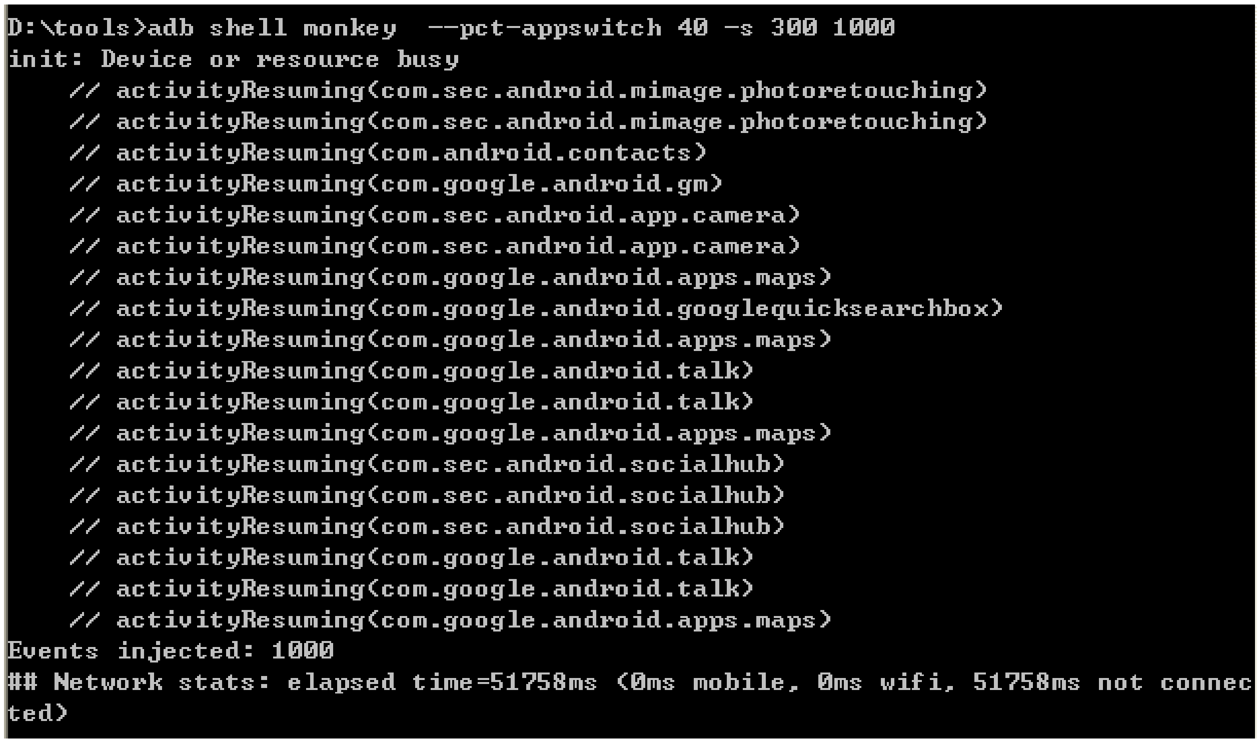
Two SW platforms are tested for comparison:

- Original Android + Optimized Comp-cache
- Original Android (get from P4)

The switch application test is set as 40% in whole monkey test event.

SW Platform	1000 event	2000 event	5000 event	10000 event
Original Android + Optimized Comp-cache	Pass	Pass	Pass	Pass
Original Android	Pass	Pass	Pass	Pass

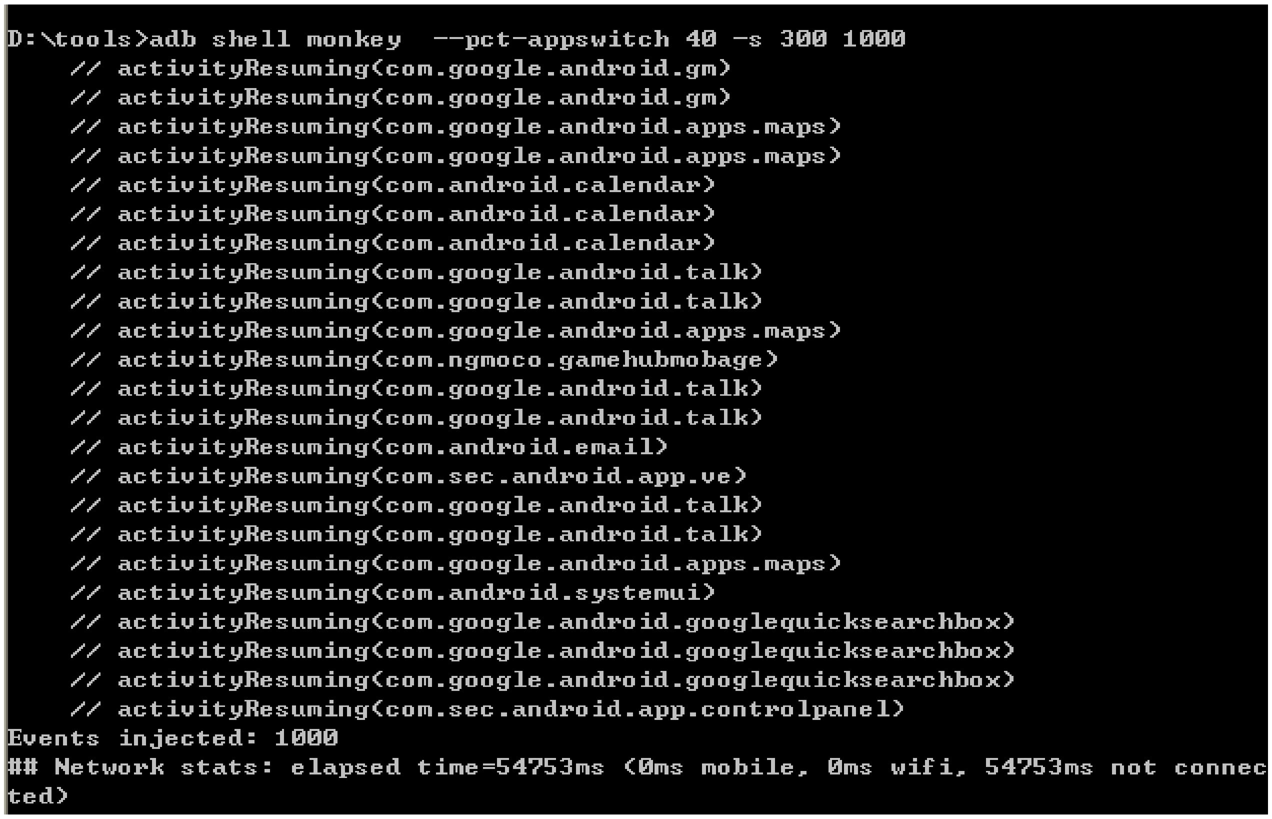
Original Android + Optimized Comp-cache



```
D:\tools>adb shell monkey  --pct-appswitch 40 -s 300 1000
init: Device or resource busy
    // activityResuming(com.sec.android.mimage.photoretouching)
    // activityResuming(com.sec.android.mimage.photoretouching)
    // activityResuming(com.android.contacts)
    // activityResuming(com.google.android.gm)
    // activityResuming(com.sec.android.app.camera)
    // activityResuming(com.sec.android.app.camera)
    // activityResuming(com.google.android.apps.maps)
    // activityResuming(com.google.android.googlequicksearchbox)
    // activityResuming(com.google.android.apps.maps)
    // activityResuming(com.google.android.talk)
    // activityResuming(com.google.android.talk)
    // activityResuming(com.google.android.apps.maps)
    // activityResuming(com.sec.android.socialhub)
    // activityResuming(com.sec.android.socialhub)
    // activityResuming(com.sec.android.socialhub)
    // activityResuming(com.google.android.talk)
    // activityResuming(com.google.android.talk)
    // activityResuming(com.google.android.apps.maps)
Events injected: 1000
## Network stats: elapsed time=51758ms (0ms mobile, 0ms wifi, 51758ms not connec
ted)
```

Figure 4-5 Monkey Test Result (Optimized Comp-cache)

Original Android



```
D:\tools>adb shell monkey  --pct-appswitch 40 -s 300 1000
    // activityResuming(com.google.android.gm)
    // activityResuming(com.google.android.gm)
    // activityResuming(com.google.android.apps.maps)
    // activityResuming(com.google.android.apps.maps)
    // activityResuming(com.android.calendar)
    // activityResuming(com.android.calendar)
    // activityResuming(com.android.calendar)
    // activityResuming(com.google.android.talk)
    // activityResuming(com.google.android.talk)
    // activityResuming(com.google.android.apps.maps)
    // activityResuming(com.ngmoco.gamehubmobage)
    // activityResuming(com.google.android.talk)
    // activityResuming(com.google.android.talk)
    // activityResuming(com.android.email)
    // activityResuming(com.sec.android.app.ve)
    // activityResuming(com.google.android.talk)
    // activityResuming(com.google.android.talk)
    // activityResuming(com.google.android.apps.maps)
    // activityResuming(com.android.systemui)
    // activityResuming(com.google.android.googlequicksearchbox)
    // activityResuming(com.google.android.googlequicksearchbox)
    // activityResuming(com.google.android.googlequicksearchbox)
    // activityResuming(com.sec.android.app.controlpanel)
Events injected: 1000
## Network stats: elapsed time=54753ms (0ms mobile, 0ms wifi, 54753ms not connec
ted)
```

Figure 4-6 Monkey Test Result (Original Android)

From the Monkey test result, optimized Comp-cache and KSM does not bring side effect to system overall stability.

4.2 System performance test on Comcache

The performance test environment is defined as below:

HW Platform: G-I9103; (835MB user space memory)
SW Platform:

Three kinds of SW platform will be tested for the performance comparison:

- Code1: Original Android (no Comp-cache)
- Code2: Original Android + Original Comp-cache
- Code3: Original Android + Optimized Comp-cache

1st test scenarios are defined as below (Free Memory Test):

- Run same quantity process on Original Comp-cache & Optimized Comp-cache platform, compare the memory cost.
- The Ramzswap size is defined as 25% of user space size----200MB

Test result is shown as following:

To test the free memory performance of original Comp-cache & optimized Comp-cache; 6 processes are created, each process will cost 100MB memory. After original Comp-cache & optimized Comp-cache works, the free memory in two system are compared.

Platform	Free memory (MB)
Original Comp-cache	18.9358
Optimized Comp-cache	147.6636

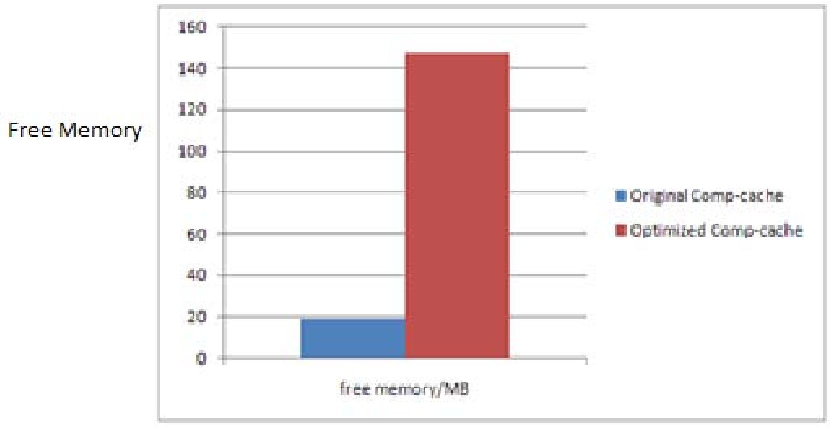


Figure 4-7 Free Memory Test Result

In this test, 129MB internal memory are saved.

Analysis of the test result is as following:

After 6 processes is running, they will occupy 600MB, and with other basic process in kernel occupying some memory, so,

- In original Comp-cache, system reaches low memory status; at this moment, if system request more pages, original Comp-cache will start, to satisfy the system request pages;
- *In optimized Comp-cache, as compress & swap will be start during system idle status, so after running 6 processes (100MB memory for each), system still has 129MB free memory*;
- Finally, 129MB/835MB=15.44% internal memory can be saved.

2nd test scenarios are defined as below:

- "Fish demo" is chosen as the test target application, which will cost 50MB memory
- Two scenarios are tested:
  - ✓ Normal status (system is in idle), there is enough memory in system in this scenario;
    - a) System free memory size (set the starting free size is 80MB);
    - b) Time cost to start a new process (test application, will request 30MB memory);
    - c) When test application is running, press "home" key, and restart it; check the restart time cost;
  - ✓ Low memory status, the free memory is lower (about 20MB);
    - a) System free memory size (set the starting free size is 20MB);
    - b) Time cost to start a new process (test application, will request 30MB memory);
    - c) When test application is running, press "home" key, and restart it; check the restart time cost;
- The Ram zswap size is defined as 25% of user space size----200MB
- Each scenario is tested for 5 times, and the average value will be considered as the test result.

Test result is shown as following:

Scenario	Test Item	No Comp-cache	Original Comp-cache	Optimized Comp-cache
Normal status	Free memory ①	80MB	80MB	80MB/163.7MB ④
	Start new process time ②	2080ms	2086ms	2083ms
	Restart process time	267ms	264ms	304ms
Low memory status	Free memory (Initial time)	20.5MB	20.5MB	20.5MB/104.3MB
	Free memory (time at the beginning of new process start)	90MB	70.2MB	104.3MB
	Free memory (After new process totally starts)	46.1MB	22.6MB	57.4MB
	Start new process time	2315ms	2730ms	2014ms
	Restart process time	2061ms	298ms	306ms

Note:

- Free memory: current system free memory;
- Start a new process time: The time cost that system starts a new process;
- Restart process time: for the test application, press home key, and then restart it, the time cost of restart the test application;
- 80MB/163.7MB: the initial status is 80MB, after screen off, optimized Comp-cache works in system idle status, and the free memory becomes 163.7MB;
- Free memory (initial time): the initial status is 20.5MB for each platform, after the "fish demo" (process costs 30MB and other related service 20MB, totally costs 50MB) starts, system will be in low memory status
- Free memory (time at the beginning of new process starts): at this point of time, new process requests free memory to system, and each platform will do different action. (No Comp-cache will start LMK to kill process; Original Comp-cache will begin to compress/swap in-active pages; Optimized Comp-cache has enough free memory, so it will just allocate memory for new process) This line show the free memory status at this point of time.
- Free memory (after new process totally starts): After new process totally starts, this process will cost some memory. This line show the free memory at this point of time.
- In No Comp-cache, when system has only 20.5MB, the "fish demo" starts (it will cost totally 50MB), system is in low memory status; system will firstly call LMK to kill a process to free enough memory (in this test, the "Angry bot" is killed), 70MB memory is free by LMK, so at this moment, free memory is 90MB. (Attention, in this case, a process is killed, but for other 2 platform, no process will be killed.) ;

  And after "fish demo" totally start, as it cost around 50MB, now, the system has only 46.1MB.
- In Original Comp-cache, when system has only 20.5MB, the "fish demo" starts (it will

cost totally 50MB), system is in low memory status; original Comp-cache starts; it will compress/swap pages according to how many pages system requests; totally 50MB will be free by original Comp-cache during its compress/swap, so at this point of time, system free memory is around 70.2MB;

And after "fish demo" totally start, as it cost around 50MB, now, the system has only 22.6 MB.

– At the initial time, for Optimized Comp-cache, the free memory is 20.5MB, and then, press power key, optimized Comp-cache will start, system will have totally 104.3MB. So, when "fish demo" starts, system has enough free memory to run it. And after "fish demo" totally start, as it cost around 50MB, now, the system has only 57.4 MB.

Free memory compare:

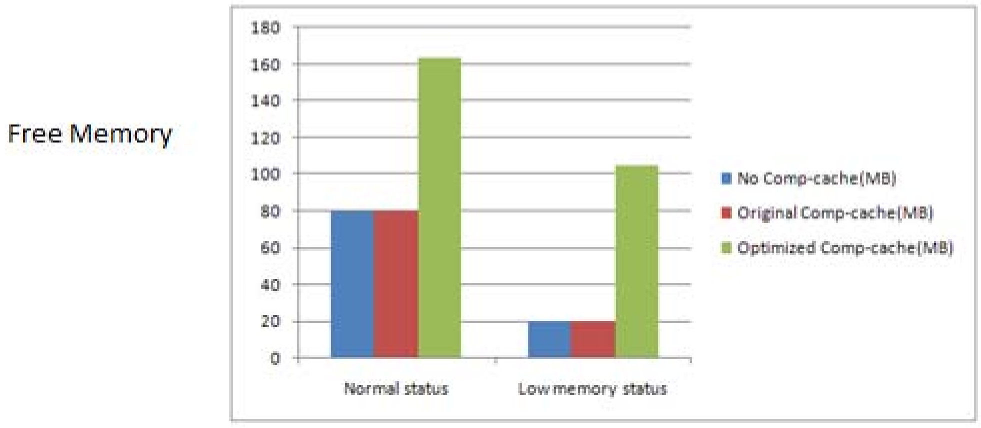


Figure 4-8 Free Memory Result

Time cost of start a new process:

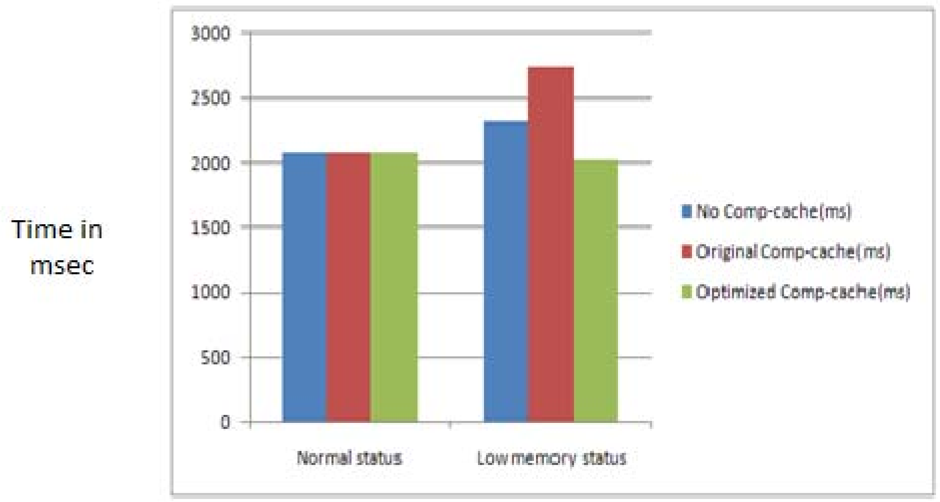


Figure 4-9 Time Cost of Starting New Process

Time cost of restart a process:

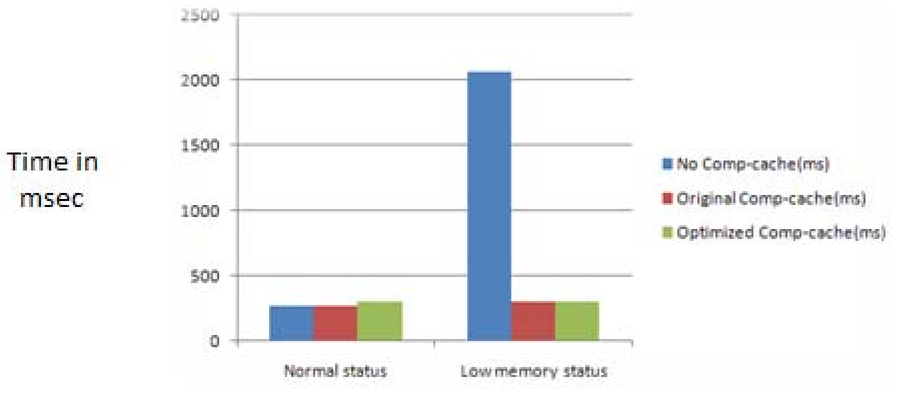


Figure 4-10 Time Cost of Restart Process

Analysis of the test result is as following:

In normal status (system is idle),

- Free memory:
  (Optimized Comp-cache(163.7MB)>Original Comp-cache(80MB)=No Comp-cache(80MB))
  After screen off, Optimized Comp-cache solution has more free memory; and at this status, no Comp-cache solution and original Comp-cache solution has same free memory, less than Optimized Comp-cache;
- Start new process:
  (Optimized Comp-cache(2083ms)=Original Comp-cache(2086ms)= o Comp-cache(2080ms))
  As the free memory is enough in all 3 platform (80MB, 80MB, 163.7MB), CPU is in idle status, so, the time cost to start a new process is almost same.
- Restart process:
  (Optimized Comp-cache(304ms)>Original Comp-cache(264ms)=No Comp-cache(267ms))
  For no Comp-cache & original Comp-cache, at this status, the test application's all pages

are located in internal memory, without any compressed page, these two will restart applications very quickly, almost same time; but for optimized Comp-cache, at this status, almost all the test application’s pages are compress & swap to Ramzswap, to restart it will cost extra decompress & swap time, so the restart time in optimized Comp-cache case, will be larger.

In low memory status (system free memory left only 20MB),

- Free memory:
  After screen off, Optimized Comp-cache solution has more free memory; and at this status, no Comp-cache solution and original Comp-cache solution has same free memory, less than Optimized Comp-cache;
- Start new process,
  (Original Comp-cache(2730ms)>No Comp-cache(2315ms)>Optimized Comp-cache(2014))
  - ✓ No Comp-cache: when system is in low memory status, to start a new process, need more free memory, so, system need to firstly do LMK, kill other not important process to free memory, then, finish starting a new process. It will cost more time. As kill a process and free corresponding memory will not cost too much time, No Comp-cache case will cost less time than Original Comp-cache;
  - ✓ Original Comp-cache: when system is in low memory status, to start a new process, need more free memory, system need to do Comp-cache firstly, free some memory, then, finish starting a new process. It will cost more time.
  - ✓ Optimized Comp-cache: in this situation, there is enough memory in system, system will start a new process as normally. Optimized Comp-cache case will cost less time than other two cases.

- Restart process:
  (No Comp-cache(2061ms)>Optimized Comp-cache(306ms)=Original Comp-cache(298ms))
  For this case, system will firstly start other process, and then check the test process

- ✓ No Comp-cache: in low memory status, start other process, system will firstly start LMK, the test application will be killed by LMK, then, restart the test application, it will cost much time (same as start a new one, as the test application has been killed)
- ✓ Original Comp-cache: in low memory status, after press “home” key, the test application will be compress & swap, to restart it, it will cost time to decompress & swap some page (but as it is only restart, it will not cost much time)
- ✓ Optimized Comp-cache: in this status, system still has enough memory, the time cost is focus on decompress & swap all physical pages of the test process. As it is restarted, not all the pages are needed, it’s time cost is only a little bit large than the original Comp-cache.

Conclusion:

- – Optimized Comp-cache makes more free memory for whole Android platform
  - ✓ In 835MB user space internal memory, 15.44% memory can be saved (from the test result);
  - ✓ In another way, we can keep more processes exist in memory, for a even better system performance; it also means that, optimized Comp-cache can delay the system low memory status coming as later as possible.
- – At the same time, enabling optimized Comp-cache also will not bring bad side effect to whole system stability performance.

Result And Analysis of KSM

4.3 CPU Time for Scanning Page on KSM

The modified KSM scheme is tested with all the flavors of the KSM. For this 10 processes each consuming memory from 60~80MB is started. In KSM pages are compared to find the same pages, and cost per page in term of time is calculated using the pages_to_scan and time taken.

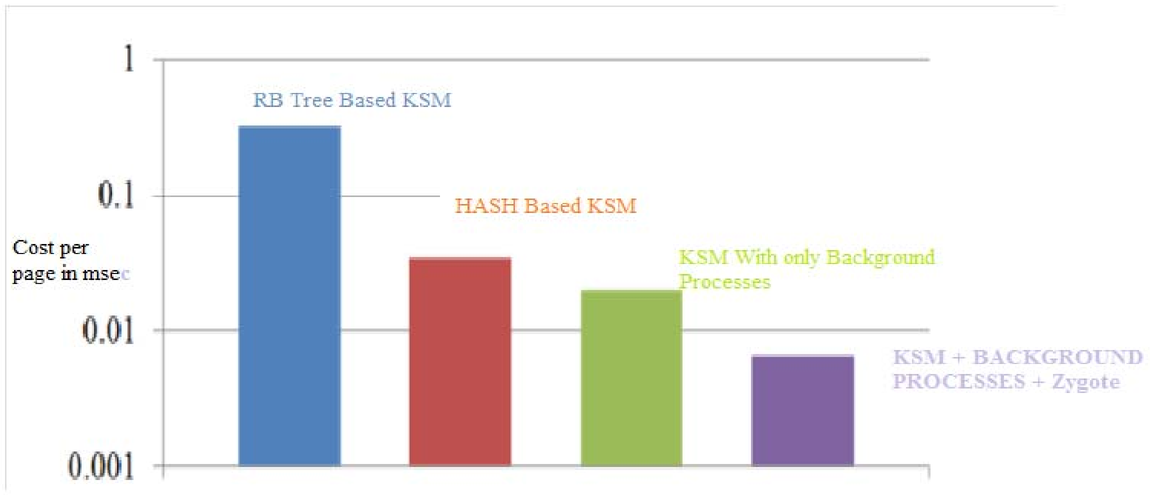


Figure 4-11 CPU Cost Per Page

Analysis of the above result

When the KSM is applied only on the back ground processes the improvement is far batter than the normal KSM. Next point of analysis required do different amount of memory is freed when with each of the scheme. This is done below.

4.4 Memory Saving with KSM various schemes

Below we have taken the statistics of the free memory in each of the case and it is observed that memory saving is all most same in each of the case.

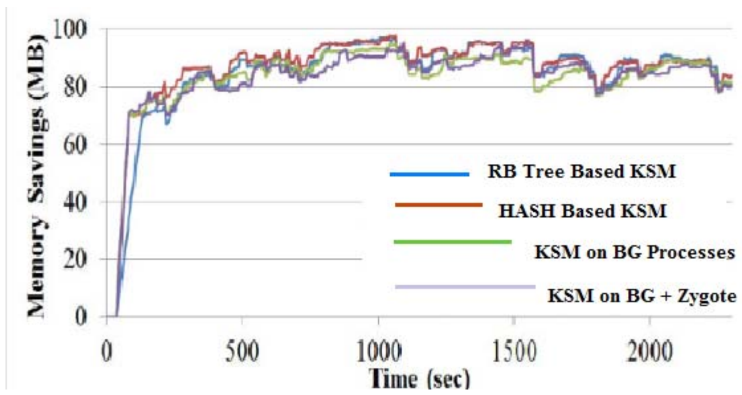


Figure 4-12 KSM Free Memory

So far we have analyzed the Com Cache and KSM, separately and drawn the various charts of the improvements. Both of the techniques we mentioned analyzed, modified and evaluated and integrated with modified LMK ultimately improves the system RAM. When the system is RAM is improved situation of memory pressure is batter. This is proved form the below chart which evaluates the calling frequency of LMK with both algorithms.

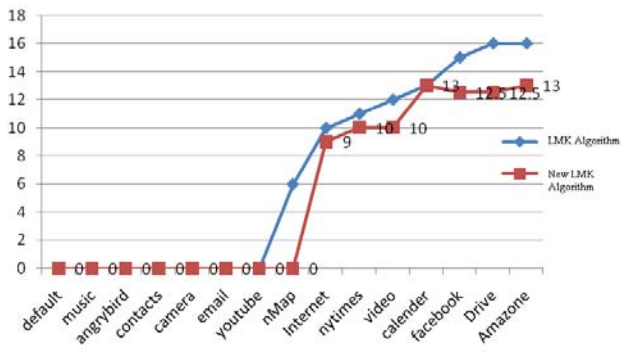


Figure 4-13 LMK Frequency

Chapter 5

Conclusion and Future Work

Modern devices with no swap space results in killing of process at low memory situations which was the case in Android. LMK is used in Android to kill the processes at low memory situation. In this thesis existing LMK is improved using in kernel compression techniques such as KSM and Com Cache. Using part of a RAM as swap space will increase the effective memory usage of main memory and it avoids killing of process to some extend at low memory situations. Modern embedded system where flash is used as secondary storage will suffers from performance when conventional virtual memory system is used. Hence integration of in kernel memory compression techniques such as Com Cache to increase the efficiency. Com Cache integration with the LMK is solved by changing the data structure and changing the make files. Calling the LMK at system idle time and when system is under memory pressure is successfully changed. All the above changes results in the higher RAM availability to the system. Other approach of keeping unique pages in case of duplicate pages ie KSM suffers from high CPU usage is solved with the proper selection of input processes. The similarity page ratio is improved with the selection of processes originated from the same ancestors.

For a mobile phone user the time taken to launch an application ie initial entry and re-entry time of the application are important, both are affected by the available RAM. This improved RAM availability has increased the system performance and Degree of Multi-programming. With this improvement the Smart Phone user feels improvement in usability and can open more applications simultaneously.

In this improved LMK incorporated two techniques, available. Another good idea is to improve on dedicated memory allocation like in case of Camera and Decoders separate memory is reserved dedicatedly. This dedicated memory allocation wastes lot of RAM particular in the situation when all the functions for which memory is reserved is not used simultaneously. Shared memory pool and changes in the memory management scheme can further improve the available RAM for the system.

References

[1]. Robert Love , Linux Kernel Development

[2]. Daniel P. Bovet, Understanding the Linux Kernel

[3]. Marko, Marakana Android Internals

[4]. Goldwyn Rodrigues, Taming the OOM Killer, lwn.net/Articles/317814/

[5]. Anatomy of Android , http://anatomyofandroid.com/2013/10/15/zygote/

[6]. https://www.ibm.com/developerworks/linux/library/l-kernel-shared-memory/

[7]. Nitin Gupta Anderson Farias Briglia, Allan Bezerra. Evaluating effects of cache memory compression on embedded systems. *2007 Linux Symposium*, 2007.

[8]. Jennings, The zswap compressed swap cache, http://lwn.net/Articles/537422/

[9]. A. Arcangeli, I. Eidus and C. Wright, "Increasing memory density by using KSM", In Proceedings of the Linux Symposium, (2009) July 13-17; Montreal, Canada

[10]. Red Hat Chapter 8 , https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Virtualization_Administration_Guide/chap-KSM.html

[11]. Senthilkumaran R, Hash Based KSM, http://www.cse.iitb.ac.in/synerg/lib/exe/fetch.php?media=public:students:senthil:report.pdf

[12]. https://www.kernel.org/

[13]. Activity Life cycle. http://developer.android.com/reference/android/app/activity.html, last accessed March 2013

[14]. Ramani Yellapragada. *Linux Memory Management*. 2003.

[15]. M.I. Vuskovic. Virtual memory in operating system lecture notes, last accessed October 2012.