Declaration

I declare that this written submission represents my ideas in my own words and where other’s ideas or words have been included, I have adequately cited and referenced the original sources. I also declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact/source in my submission. I understand that any violation of the above will be cause for disciplinary action by the Institute and can also evoke penal action from the sources which have thus not been properly cited or from whom proper permission has not been taken when needed.

Subbanjaneyulu Reddy
(113050021)

Place: IIT Bombay, Mumbai
Date: 25th June, 2013
Abstract

Android is a software platform and operating system for mobile devices, based on the Linux kernel. Memory is a scarce resource in embedded systems like Android. The complexities and resource demands of modern embedded systems such as Android smartphones are constantly increasing. Increasing memory often increases packaging costs, cooling costs, size, complexities and power consumption. At low memory situations Operating System swap some of the pages to swap device, usually secondary storage to find some free memory in RAM. Android does not have swapspace as the secondary storage is Flash memory, which suffers from wear leveling property. When there is a scarcity of memory, Android Low Memory Killer kills some of the processes to free the memory. Compressed RAM is a technique where pages are compressed and stored in the RAM itself. Objective of the Compressed RAM project is to use part of the RAM as swap device. The pages swapped to this device are compressed and stored. Hence the effective memory of the device increases. This Thesis investigates traditional Compressed RAM approach and designed a new approach for Compressed RAM called as enhanced Compressed RAM. We have simulated both the approaches where we are able to double the number of pages stored in compressed area than in traditional approach.
## Contents

1 Introduction

1.1 Memory Management in OS

1.1.1 Address space

1.1.2 Contiguous memory allocation

1.1.3 Segmentation

1.1.4 Paging

1.1.5 Translation Look Ahead Buffer

1.2 Page Replacement

1.2.1 Global replacement Strategy

1.2.2 Local replacement Strategy

1.2.3 FIFO Replacement

1.2.4 Optimal algorithm

1.2.5 Least Recently Used Algorithm

1.2.6 Approximation to LRU :Second chance Algorithm

1.2.7 Enhanced second chance algorithm

1.3 Linux Page Frame Reclaiming Algorithm

1.4 Organization of Thesis

2 Process Management in Android

2.1 Process Management

2.1.1 Dalvik Virtual Machine

2.1.2 Application

2.1.3 Categories of Process

2.2 Activity Life Cycle

2.3 States of Activity

2.3.1 Active

2.3.2 Paused

2.3.3 Stopped
## 3 Literature Survey

3.1 Swap compression .................................................. 19
   3.1.1 Read/Write path ............................................. 20
   3.1.2 Swapping In .................................................. 22
   3.1.3 Swapping Out ................................................. 22
   3.1.4 Cache-cleaning algorithm ................................... 22
3.2 Virtual memory in system with Flash memory as secondary storage ... 23
   3.2.1 Flash Memory .................................................. 23
   3.2.2 Sub paging ..................................................... 23
   3.2.3 Hot cache ..................................................... 24
3.3 Online Memory compression for Embedded systems ..................... 25
   3.3.1 Code Compression ............................................. 26
   3.3.2 Data Compression ............................................. 26
   3.3.3 CRAMES ....................................................... 26
   3.3.4 Virtual Memory Swapping .................................... 26
   3.3.5 Request Handling ............................................. 27
3.4 Memory Allocators .................................................. 27

## 4 Compression Algorithms

4.1 Introduction ....................................................... 31
4.2 Results ............................................................ 33
   4.2.1 Compression Ratio ......................................... 33
   4.2.2 Decompression Time ....................................... 34
   4.2.3 Compression Time ........................................... 34
   4.2.4 Good Compressible Pages .................................. 35
   4.2.5 Incompressible Pages ....................................... 35
   4.2.6 Statistics of Zlib ........................................... 36
   4.2.7 Conclusions ................................................. 36

## 5 Design of Compressed RAM

5.1 CRAMES Approach .................................................. 39
   5.1.1 Observations ................................................ 40
5.2 Proposed System .................................................. 40
   5.2.1 Pages that should be targeted ............................ 40
   5.2.2 Properties of pages ....................................... 41
   5.2.3 Results ...................................................... 44

## 6 Conclusions


## 7 Further work
## List of Figures

1.1 User process and operating system in main memory ........................................ 2
1.2 Address translation From virtual to physical address ...................................... 2
1.3 contiguous allocation of memory to process .................................................... 2
1.4 Memory allocation to process using segmentation ............................................. 4
1.5 Address Translation using paging ................................................................. 5
1.6 System calls invoked when the pages are reclaimed in Linux .......................... 8
1.7 Transitions among the two LRU lists ............................................................ 10
1.8 Linux page frame reclaiming overview .......................................................... 10
2.1 State of Process ......................................................................................... 14
2.2 Activity Life cycle ...................................................................................... 16
3.1 Conceptual vision of the compression and cache mechanism .......................... 20
3.2 Graphical example of the two kinds of read hits ........................................... 21
3.3 Compression and Cache mechanism with two different swapping paths ........ 22
3.4 Time-Frequency breaks the intra page locality .............................................. 24
3.5 Mechanism of Two level LRU .................................................................... 25
3.6 Logical structure of swapping mechanism on compressed RAM device ....... 27
3.7 Handling request in CRAMES ..................................................................... 28
3.8 TLSF data structure .................................................................................... 30
4.1 Average compression ratio ........................................................................... 34
4.2 Average decompression time ......................................................................... 34
4.3 Average compression time ........................................................................... 35
4.4 Number of pages with compression ratio \( \geq 2 \) ........................................... 35
4.5 Number of pages with compression ratio \(< 1 \) .............................................. 36
4.6 Percentage of pages with \( 1/\text{compression-ratio} \) in the range .................... 36
5.1 Free memory with respective to time line .................................................... 45
5.2 Number of pages stored in compressed area ............................................... 46
5.3 Number of compressions and decompression .............................................. 46
# List of Tables

1.1 Classification of pages .................................................. 8
2.1 Description of each state of process.............................. 16
4.1 Comparison of various Compression algorithms............ 32
Chapter 1

Introduction

In this chapter we will see the memory management of Operating Systems, different ways the OS allocates memory to process, the way Linux reclaims main memory pages so that it always have some free memory to run process.

1.1 Memory Management in OS

This section will introduce basics of memory management like physical address, virtual address, different memory allocation techniques to process, address translation and protection.

1.1.1 Address space

The memory space utilized by the process for running is called as Address space[1][7]. Each process can have address space as large as the virtual address. The abstraction provided by the address space allows each process to have same set of virtual address which are logically distinct. Virtual memory allows the process to have address space more than that of the physical memory. This allows the machine to run a process which has more address space than the physical memory. In multi programming environment, where multiple processes exist in main memory at same time requires memory protection as one process should not access the memory of some other process.

The logical address is different from physical address. The process resides in physical address space, but it uses logical address for its computation by using dynamic address translation as shown is figure 1.2.

1.1.2 Contiguous memory allocation

The entire process address space is allocated contiguous in main memory[7].
Translation

The translation from logical address to physical address is done by MMU unit as shown in figure 1.2. Let the base address of the process where the allocation starts be X. So any logical address, say Y of the process is computed as X+Y as its physical address by
MMU.

Protection

Whenever we need to access logical address, say Y OS computes X+Y and checks to see whether it goes beyond the range of memory allocated to this process and if so signals illegal memory access and does not allow it. This way the OS protects the process from accessing other process memory.

Advantages

• Simple in implementation, need only two register which keep track of starting address and ending address for each process

• Low overhead

Disadvantage

• Does not allow the process with addresses space more than physical memory to be executed.

• This method of allocation leads to external memory fragmentation.

1.1.3 Segmentation

The logical address of the process is divided into segments like code segment, data segment, etc. Each segment is allocated contiguously in the physical memory. Virtual address is identified by segment id and offset.

Protection and Translation

For each segment we keep track of the starting and the ending physical address. Whenever a process needs access to a logical address, say X OS computes which segment the X falls in and then computes the physical address based on starting address of the segment and check against the ending physical address.

Advantages

• Reduces external fragmentation.

• Works well for sparse address space.
Disadvantages

- A process cannot use all the virtual address because the segment size are of different size.
- It suffers from external fragmentation.

### 1.1.4 Paging

Logical Address is divided into equal sized chunks called as pages. Physical address is also divided into same sized chunks called as frames or page frames. A page can fit into any frame because both are of same size. OS keep track of which page goes into which frame by using a table called page table. Each process has its own page table. Virtual address is divided into two parts, page number and page offset.

#### Translation

Whenever a process generates a virtual address, OS obtains the page number from the virtual address and look into page table by using page number as index. If the page is accommodated in main memory then the entry in page table contains the corresponding frame number. If not, then an invalid bit is set to indicate a page fault and the OS deals with page faults by allocating memory for the page and updating the page table. Once the frame number is obtained, it is added with page offset which is a physical address.
Protection

Whenever the page of the frame is evicted from the main memory the corresponding entry in the page table is invalidated. This way it ensures protection.

Advantages

- No external fragmentation.
- Demand paging leads to virtual memory.

Disadvantages

- Internal fragmentation occurs in the last page of the process.
- Each page access results in two memory access which is slow.

1.1.5 Translation Look Ahead Buffer

Translation using paging involves two main memory access. Main memory access is expensive. TLB cache stores the page number to frame number mapping. So, when a hit in TLB occurs, only one main memory access and TLB access is required. TLB access is less expensive. When an miss occurs in TLB it leads to search in page table as mentioned in section 1.1.4.
1.2 Page Replacement

When a page fault occurs and there is no free frame then one of the pages in memory has to be replaced. The strategy to choose which page to replace is called as page replacement algorithm \[7\] [17]. Below, we mention the most common replacement algorithms and strategies for paging.

1.2.1 Global replacement Strategy

When a page is to be replaced one of the pages among all process is selected. The behavior of other process may effect the paging activity. It incurs less overhead in implementation.

1.2.2 Local replacement Strategy

When a page is to be replaced the one of the pages belonging to same process is selected to be replaced. The behavior of other process does not have impact on the paging activity. It incurs more overhead in implementation.

1.2.3 FIFO Replacement

The page which entered the main memory first is selected for replacement. It can be implemented using a queue. It suffers from Belady’s anomaly i.e. when the no of frames increases there is chance of increase in page faults.

1.2.4 Optimal algorithm

Replace the page which will not be used in near future. Can not be implemented as page references occurs at runtime. It acts as benchmark to find efficiency of other algorithms.

1.2.5 Least Recently Used Algorithm

Replace the page which is not used in near past. Does not suffer from Belady’s anomaly.

Time stamp implementation of LRU

Whenever an page is referred update in page table its time of access with current time(counter). When ever a page is to be replaced the page with least page time access is to be searched. Searching introduces substantial time overhead.
Stack based implementation of LRU

Page numbers are stored in special stack. When an page is referred it is removed from stack if exists in stack and placed it on top of the stack. If the referred page is not in the stack it is placed on top of the stack. So the most recently accessed page always stay on the top of the stack and least recently accessed page are in the bottom of the stack. Hence the page are selected from bottom of the stack for replacement. Both the above methods incurs lot of overhead in implementation which is propositional to length of the stack i.e number of pages allocated in main memory.

1.2.6 Approximation to LRU :Second chance Algorithm

It is implemented using circular buffer. Each entry in the buffer holds the page number and reference bit. A pointer is used which points to the slot in the circular buffer which has to be next examined. Whenever a page is referred the reference bit is set as 1. When a page fault occurs the slot pointed by the pointer is examined and if the reference bit is 1 it is given chance by setting it as 0 and examining the next slot. If the examined slot has reference bit as 0 it is chosen as the page to be replaced. As the page which is skipped as marked as 0 it will not lead to an infinite cycle when all slots has reference bit as 1.

1.2.7 Enhanced second chance algorithm

It introduces an extra bit called as dirty bit. When the page content is changed the dirty bit of the page is marked as 1. When a page is to be replaced the selection is based on combination of dirty bit and reference bit.

The increasing order of priority of the page to be selected is:

R D
0 0
0 1
1 0
1 1

1.3 Linux Page Frame Reclaiming Algorithm

The Linux Page Frame Reclaiming Algorithm(LFRA)[6] has to make sure that there is some free memory in the system at any point of time. Page are classified into four classes and the following table 1.1 shows the same.

Mapped page is the one which is part of the file. It corresponds to a particular block in the file system. Whereas, anonymous pages which do not correspond to any file is part of process space like stack, heap etc.
<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unreclaimable</td>
<td>Locked pages, Kernel mode stacks, free pages,</td>
<td>Impossible</td>
</tr>
<tr>
<td></td>
<td>reserved paged</td>
<td></td>
</tr>
<tr>
<td>Swappabe</td>
<td>Anonymous user mode pages, mapped pages of tmpfs file system</td>
<td>Write to swap area</td>
</tr>
<tr>
<td>Syncable</td>
<td>Mapped pages which are part of file</td>
<td>Write to file</td>
</tr>
<tr>
<td>Discarded</td>
<td>Unused pages</td>
<td>Nothing to be done</td>
</tr>
</tbody>
</table>

Table 1.1: Classification of pages

The LFRA is invoked at low memory situations, hibernate mode of the system, periodically to make sure memory is available at all the time. The LFRA tries to reclaim the pages that are not associated to any process because they do not need any changes to page table. When a shared page is reclaimed the page table of all the process that share it should be changed. It tries to reclaim the page that haven’t been referred recently. The following figure 1.6 shows the system calls invoked in Linux while reclaiming the pages. System maintain two LRU lists, the active list and inactive list. when a page is not referred recently the page is moved to inactive list. When a page is referred for the first time it the PG_referenced flag is set. When a page with PG_referenced bit set is accessed it is moved to active list. when the page with PG_referenced bit set is not accessed for long time the PG_referenced bit is reset. The following figure 1.7 shows the transitions between active and inactive list. When the pages is to be reclaimed the pages in inactive

Figure 1.6: System calls invoked when the pages are reclaimed in Linux

Figure 1.7: Transition between active and inactive list.
Swapping is the process of swapping out the anonymous pages to disk. By doing this it extends the amount of memory it can access at the expense of access speed. It allows to make frames free so that it can be used by other process. As described in table 1.1 swapping applies only to anonymous pages of the process, modified pages belonging to private memory mapping of a process, pages belonging to an IPC (Inter Process Communication) shared memory region. Swapping out mechanism tries to steal pages from the process having largest number of pages in RAM [5]. But it makes a choice among the pages of the selected process based on access time. Some machines have hardware support like incrementing the count when a page is accessed and some do not have. LRU lists can be maintained based on such support or can use access time parameter in page table entry. When the system has no free and no reclaimable memory the system is in trouble and Out of memory killer is invoked which kills some of the process. To avoid trashing one process at a time holds the swap token. Pages belonging to the swap token owner are never reclaimed. kswapd is the kernel thread which swaps the swap pages (anonymous pages). Based on swappniess value the kernel selects either syncable pages or swapable pages. Figure 1.8 gives an overview of the algorithm.
Figure 1.8: Linux page frame reclaiming overview
1.4 Organization of Thesis

This Thesis investigates traditional Compressed RAM approach and designed a new approach for Compressed RAM called as enhanced Compressed RAM. We have simulated both the approaches where we are able to double the number of pages stored in compressed area than in traditional approach. Chapter 1 explains memory managed in traditional desktop OS, Page frame reclaiming algorithm in Linux and page replacement algorithms. Chapter 2 give details about process management in Android, Activity life cycle, classification of process, priorities of process while killing them. Chapter 3 explains some techniques that increase the performance and memory utilization of the system like use of cache for swap space, use of compressed RAM as swap device, concept of sub paging in virtual memory system. Chapter 4 compares some of the well known compression algorithms. Chapter 5 details about the traditional Compressed RAM approach and the proposed Enhanced Compressed RAM approach. Chapter 6 and 7 explains the conclusion and further work respectively.
Chapter 2

Process Management in Android

In this chapter we will see the process management of Android, categories of process, activity life cycle of application etc.

2.1 Process Management

Android is Linux based OS. All the basic OS operations like I/O, memory management etc. are handled by the native stripped-down Linux kernel. Android manages application memory by using its own run time and virtual machine. The process life time is also managed by Android run time. Android stops and kills the process as necessary to allocate resources to high priority applications.

2.1.1 Dalvik Virtual Machine

Android uses Dalvik virtual machine. Each Android application runs in a separate process within its own Dalvik instance\[12\], giving all responsibility for memory and process management to the Android run time, which stops and kills processes as necessary to manage resources.

Dalvik is a register-based virtual machine that is been optimized to ensure that a device can run multiple instances efficiently. It relies on the Linux kernel for threading and low-level memory management.

Traditionally all the mobiles which are developed on Java uses Java virtual machine like Java Mobile Edition. But Android uses more optimized virtual machine to ensure that multiple instances run efficiently on a single device. Dalvik uses the device underlying Linux kernel to handle low-level functionality like security, threading, process and memory management. All application are in .dex format. Which is an optimized format to ensure low memory foot print. The Dalvik virtual machine executes .dex executable files. The applications are written in Java and transformed into .dex by using tools in SDK.
2.1.2 Application

Application will have components. Each component will have priority. The priority of the application is the highest of priorities of all the components of the application. The application with low priority is killed at low memory situations. Process priority is also effected by interprocess dependencies. When an application depends on service provided by second application. The second application should have priority at least equal to the priority of first application. Android tries to keep the application in memory as long as it does not need resources.

2.1.3 Categories of Process

- Active process: The application with components which are interacting with the user are classified as active process. The android tries to kill them at the last. The android tries not to kill so that it can interact with the user.

  - Active process contains:
    - Activities in active state. An activity is said to be active if they are in foreground and interacting to user events.
    - Activities, services or broadcast receivers that are currently executing an onReceive event handler.
    - Services that are executing an onStart, onCreate, onDestroy event handler.

- Visible process: The inactive process that are not in foreground or not responding to user events but still visible are called as Visible process. They will be killed at last to allow the active process to continue.

- Started service process: Processes hosting Services that have been started. Services support ongoing processing that should continue without a visible interface. As
Services do not interact directly with the user, they receive a slightly lower priority than visible activities. They are still considered to be foreground processes and won’t be killed unless resources are needed for active or visible processes.

- **Background Process**: Process that have activities that are not visible and do not have any services started are classified as Background process. Android tries to kill this process whenever resources are less. Android uses last seen first kill pattern to kill the process[12].

- **Empty Process**: To increase the efficiency Android retains the application in memory even after its life time so that it will reduce the startup time for the application. The process which reside in memory even after there life time are called as Empty process. This process are given more priority to kill them. Android allows maximum of 20 such empty process. Android does this so that the load time for the application when it restarts is zero and hence have less response time.

### 2.2 Activity Life Cycle

An activity is a single, focused thing that the user can do. All the activities interact with the user. Activity Stack[4] is used by Android to maintain the activities. When an new activity is started it is placed on top of the stack and the activities which lie below it will not come to foreground until the new activity exits.

### 2.3 States of Activity

#### 2.3.1 Active

When the activity is on the top of the stack i.e in the foreground then the activity is said to be in Active state.

#### 2.3.2 Paused

When an activity is visible but lost its focus then the activity is said to be in Pause state. Paused activity is alive but may be killed by the Low Memory killer at low memory situations

#### 2.3.3 Stopped

When an activity is obscured by another activity then the activity is said to be in Stopped state. It is not visible to the user and it will retains the state information. It is more often killed by the Low Memory Killer at low memory situations.
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
<th>killable</th>
<th>Next</th>
</tr>
</thead>
<tbody>
<tr>
<td>onCreate()</td>
<td>Called when the activity is first created</td>
<td>No</td>
<td>onStart()</td>
</tr>
<tr>
<td>onRestart()</td>
<td>Called after your activity has been stopped, prior to it being started again</td>
<td>No</td>
<td>onStart()</td>
</tr>
<tr>
<td>onStart()</td>
<td>Called when the activity is becoming visible to the user</td>
<td>No</td>
<td>onResume() or onStop()</td>
</tr>
<tr>
<td>onResume()</td>
<td>Called when the activity will start interacting with the user</td>
<td>No</td>
<td>onPause()</td>
</tr>
<tr>
<td>onPause()</td>
<td>Called when the system is about to start resuming a previous activity</td>
<td>Yes</td>
<td>onResume() or onStop()</td>
</tr>
<tr>
<td>onStop()</td>
<td>Called when the activity is no longer visible to the user</td>
<td>Yes</td>
<td>onRestart() or onDestroy()</td>
</tr>
<tr>
<td>onDestroy()</td>
<td>The final call you receive before your activity is destroyed</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2.1: Description of each state of process[11]

An activity which is paused or stopped is killed at low memory situations by Android framework. When they are displayed to the user it must be completely restarted and resorted to previous state. When ever an application is about to kill by Android it will send an signal to that application so that it can write some current state to the disk and it can resume to that state when restarted.

![Activity Life cycle](image)

Figure 2.2: Activity Life cycle [4]

The entire lifetime of an activity happens between the first call to onCreate() through to a single final call to onDestroy(). An activity will do all setup of ”global” state with onCreate(), and release all remaining resources with onDestroy(). For example, if it has a thread running in the background to download data from the network, it may create that thread with onCreate() and then stop the thread with onDestroy().
The visible lifetime of an activity happens between a call to `onStart()` and corresponding call to `onStop()`. During this time the user can see the activity on screen, though it may not be in the foreground and interacting with the user. The `onStart()` and `onStop()` methods can be called multiple times, as the activity becomes visible and hidden to the user.

The foreground lifetime of an activity happens between a call to `onResume()` and corresponding call to `onPause()`. During this time the activity is in front of all other activities and interacting with the user. An activity can frequently go between the resumed and paused states. Table 2.1 describes each state and figure 2.2 indicates the transition between the states.
Chapter 3

Literature Survey

This chapter gives the details of some techniques which increases the effective memory of the device like Compressed RAM, memory allocators that can be used in Compressed RAM. It also gives insight of techniques which increase the performance of the system like swap compression, sub paging etc..

3.1 Swap compression

The performance of memory-intensive applications tends to be poor due to the high overhead added by the swapping mechanism. Even in the multi-programming system when all the applications need to use swap space to run the applications may incur high overhead. The performance may be improved by using cache and by compressing the pages.

The main problem with the swapping mechanism is the effect it has on the performance of the applications that use it. Every time the application needs to move data from/to the swapping device, it needs to access the disk. As disks are slow devices, the overhead of using them degrades the performance of applications.

- Memory page: Memory is divided into portions of fixed size. These portions of memory are called pages.

- Cache buffer: Cache is introduced to keep swapped pages. This cache is made of buffers, and each of them may contain one or more memory pages. The size of a buffer has to be at least equal to the page size.

- Disk block: The contents of cache buffer is stored into the disk. A Disk block is the portion of the disk where the contents of the cache buffer is stored. The size of the disk block is multiple of size of the a cache buffer.
• Swap out\cite{3}: As mentioned in section 1.3 Whenever the system need an free page and all the page are in use then the system send some pages to swap device. This process is called as Swap out.

• Swap in: The process of bringing the pages from Swapping device to physical memory is called as Swap In.

• Page-cache\cite{1}: The pages of the file are called as page cache pages. They are also called as file backed pages

• Swap-cache: The pages that are not part of the file are called as swap cache pages. They are also called as Anonymous pages.

Whenever a page is send to the disk, it is compressed and placed in cache buffer. When a page is needed by swapping module it checks in cache if not found, it is brought from disk back to cache then uncompressed and given to swapping module.

3.1.1 Read/Write path

Whenever a read hit is found it is because of two cases. In the first case when an read request is made the requested page is still in the cache because the page has been recently swapped out and the system has not sent the buffer to the disk yet or the cache buffer
is not used to place other pages. This type of hits are called as READ HIT DUE TO WRITE. In the second case when the page is in the cache because it is not recently swapped out, but it is in the cache because it is brought along with some previous page request. This type of hit is called as READ HIT DUE TO READ. These kind of hits are also called as prefetching hits. When a page is requested, the system prefetch all the pages in the that disk block and if any of these pages is requested later on a READ HIT DUE TO READ occurs.

Figure 3.2: Graphical example of the two kinds of read hits

- The figure 3.2 represent the initial scenario where the compressed cache is clean and is not yet used by the application
- Then the application swap out page pk, which is compressed and stored in the cache.
- When an application needs page pl. The system reads the disk block that holds the requested page and place it in cache as it is not in the cache. In the above example the disk block contains two pages and hence both of them are brought to the cache.
- When an application need pk, it does not need to go to disk as it is available in the cache itself. This is called as READ HIT DUE TO WRITE. When an application need page pm, again it results in hit called as READ HIT DUE TO READ.

Results shows that the Hits due to read are very less[3]. The reason is the order in which we swap out the pages is not same as the order in which we swap in the pages. Hence the caching the read pages from disk does not make any significant improvement
in performance. This idea leads that the cache should only be used as a write buffer and the swapped-in buffers should not be placed in the cache. Hence the final prototype does have two different path for read and write and is shown in the figure 3.3.

3.1.2 Swapping In

When the swapping module needs to swap in the page, it first examines the compressed cache and if found returns it by decompressing it. If the page is not in the cache, the system reads the block containing the page, decompress the page and send to swapping module. The block is not placed in the cache.

3.1.3 Swapping Out

Whenever the system need to swap out the page it compress the page and finds the free space in the cache to accommodate the compressed page. Here some algorithms like first-fit, best-fit, worst-fit can be used. If the system can not find such free space it cleans the buffer by placing the buffer on to the disk. Once the cache is cleaned it finds the free space to accommodate the compressed page and store it in the cache.

3.1.4 Cache-cleaning algorithm

To reuse the buffer to place new pages the pages already in the cache are to be stored onto the disk. This process is called as Cache-cleaning algorithm\[3\]. When ever enough amount of space in cache is full the system finds the continuous space in the swap device and are written contiguously in the disk. This increases the performance of the write operation significantly because of only one seek time.
3.2 Virtual memory in system with Flash memory as secondary storage

Operating system implements virtual memory system to allow process with virtual address space more than physical address space to be executed. But the Virtual memory system is designed keeping in view magnetic disc as secondary storage. Now a days Flash memory acts as secondary storage for many of the systems like in Android. The conventional Virtual memory system degrades the system performance and decreases the life of secondary storage when Flash memory is used as secondary storage.

3.2.1 Flash Memory

Flash memory works little different from magnetic disc. The operations allowed on Flash memory are read, write, erase [18]. Read operation is same as the read in magnetic discs. Update operation is different from than that of magnetic discs. Whenever we want to update a page on the flash memory we need to find an free page, write to that free page and invalidate the old page. The invalidated page is called as Dead page. When most of the page are invalidated i.e less free space, the system do the erase operation on the dead pages to reclaim it. This operation is called as Garbage collection. Typically the size if the page in Flash memory is 512 B and in main memory is 4KB. The block size in flash memory is 16 KB. So one page in main memory is equivalent to 8 pages in Flash memory. Flash memory has limited number of erase operations.

3.2.2 Sub paging

In conventional virtual memory system whenever a dirty page is to be written back to secondary storage it write entire page. So when the secondary storage is flash memory it results in 8 writes[16]. Usually not all portions of the main memory page are dirty. So it results in write operation of even clean flash pages which results in degrade in performance and reduce in life span of the flash memory since the write operation on the Flash is so expensive. So the granularity of write on to flash memory is to be the page size of flash memory not the page size of main memory. This leads to idea of subpaging. The main memory pages are divided into sub pages of flash memory size. Each sub page is associated with a bit called as dirty bit which keep track of whether the subpage is dirty or clean. On a page fault the subpages with dirty bit set are written to flash memory. Clean First Recently Used replacement algorithm can be used to reduce writes to flash memory by keeping the dirty pages in memory as long as possible.
3.2.3 Hot cache

To reduce writes to flash memory frequent writes are kept in SRAM called as Hotcache. Hotcache is organized as a fully associative cache\cite{16}. The block size of Hotcache is page of the flash memory. The cache management is done by using one of the three policies described below.

Here the flash block contain 16 flash pages. Each main memory page is equivalent to 8 flash pages. Consider in the case (a) where all flash pages of main memory page A and B are in same block. where as in case (b) there are in different block. when an swap out of page A and B is done. In case (a) the entire block is dead where as in case (b) block contain both dead and live pages. Time Frequency does not guarantee allocation as in case (a).

- Time-Frequency: Each write request is cached in Hot cache. The replacement is based in timestamp * writecounts. The block with smaller value is chosen as victim to replace. This policy prevents from hot page from being replaced by recently cold page. Hot page is the one which is frequently accessed. Without subpaging the entire page is written to flash memory back to back. This is called as intra-page locality. Since entire page is in same block of flash memory. Here the hot data is clustered hence it is more likely to find victim block with more number of dead pages when garbage collection is needed. The Time- Frequency does not guarantee this type of allocation as all the subpages of the page are written to flash memory back to back.

- Time-Frequency-Locality: To maintain the intra-page locality we need to force all the pages of the same virtual memory page to be placed in sequence. The virtual
number of the victim block is recorded and a counter is used to keep track of how many Hotcache blocks in the same main memory pages have been replaced. A hotCache block with same recorded virtual page number and with small timestamp * writecounts is selected as victim. Once the counter is zero the block with small timestamp * writecounts is chosen as victim then main memory page number is recorded and the counter is reseted.

- Two level LRU: Two LRU list are maintained. First level LRU list records the page considered as hot data. The second level LRU list records the page which can be candidate of hot data. When ever a page is accessed more frequently then it is likely to be hot data. Hence the data is given an entry to second level LRU list. When a page in second level LRU is accessed frequently it is promoted to First level. When the first level LRU is full, pages from First LRU are demoted to Second level LRU. When second level is full, pages are discarded. Figure 3.5 indicates the same.

3.3 Online Memory compression for Embedded systems

The memory requirements of the embedded system have grown faster then they originally anticipated by the designers by introduction of many applications like 3D games etc. At this stage the designers will have two options either redesign the system or come up with an solution that increases the effective memory capacity. With the new design by adding extra memory will not only takes time but also increases in equipment cost, cooling cost,
extra complexity in design. The basic idea with second option is to compress the memory so that the effective memory increases. Compression can be applied for both code and data.

3.3.1 Code Compression

The code is decompressed during application execution. When the application is not executed it is compressed and stored. Since the code does not change while execution the compression can be done offline. While the decompression is to be done online. Hence the compression algorithm with different time to compress and decompress is needed. The compression algorithm should take less time to decompress and can take more time to compress. Code does not execute sequentially as it contains conditional and unconditional statements. So the system should be able to uncompress the random access code. Most of the code compression algorithms are hardware based.

3.3.2 Data Compression

Code usually take smaller portion of memory and the larger position of memory is occupied by data. Code can even be placed in flash memory and can be executed directly from flash memory without bringing it to RAM. The compression technique used for compression of data is software based. The compression algorithm should work well with both compression and decompression.

3.3.3 CRAMES

Compressed RAM for Embedded System (CRAMES) is the software based compression technique. It compress data part. It increases effective memory without increasing physical memory. Here main memory is divided into two parts, one stores the uncompressed data and other stores the compressed data.

3.3.4 Virtual Memory Swapping

When the system cannot find a free page to allocate for a process the virtual memory system swap out some of the page to secondary storage. CARAMES instead of swapping out the page to secondary storage it compress the page and stores it in the RAM itself. Hence the effective memory of the RAM is increased and it can allow the process to run which have address space more than physical address. RAM is divided into two parts the compressed area and uncompress area. When there is a short of memory the system compress the page and stores in compressed area. When the process needs a page which is in compressed area the system should be able to quickly point out the
corresponding page in compressed area, uncompress it, bring it to the uncompressed area and give to the process. One or more process can read the same content, so when a such a page is compressed we keep track of the counter the number of process reading this page.

### 3.3.5 Request Handling

When an read request is made the system checks the mapping table entry and gets the actual address from the table. It even checks the compressed field in the table to determine whether a page is compressed or not. If it is compressed it gets the page size from the table and the page is decompressed and given to the process which requested the page. It also decrements the read count of the page and if it zero after read then the page can be freed to be used by some other process.

When a write request is made the system checks the usage field for 1. If it is 1 the page is compressed, an free slot from compressed area that can fit the size of the compressed page is taken and is allocated for compressed page and finally the old page is freed. Figure 3.7 shows the request handling in CRAMES.

#### 3.4 Memory Allocators

As we have discussed in CRAMES, the compressed pages are stored in compressed area. Since the compressed size of the page depends on data of the page, the compressed size is unpredictable. We need to allocate dynamic memory from compressed area. The holes created by allocating memory of different size should be handled. We have to select some
Figure 3.7: Handling request in CRAMES
allocator for that purpose.

Some of the characteristic of allocators that can be used in real time systems are:

- The allocator should have time bound execution time for allocation
- The allocator should be fast along with time bound
- Minimum external fragmentation

Some of the allocators are:

- Sequential Fits: It maintains list of free blocks using double linked lists and when a request comes it serves using some algorithms like Best-fit, First-fit etc. The blocks are connected logically by keeping header at start of each block.

- Segregated Free Lists: It maintains the set of lists where each list contain free blocks of fixed range. When a block is requested it will search from the list containing blocks of size that of the requested size. Similar way when a block is released it is added to the list corresponding to its size. Buddy systems are example of this approach. Both of above approaches are not time bounded and is propositional to the free list size.

- Bitmap Fits: Bitmap Fits maintains the set of lists where each list contain free blocks of fixed range. It uses bitmaps to keep track of empty lists. Since it maintains the presence of free lists using bitmaps and the bitmap processor instructor takes constant time to search, allocation is done in constant time. Half-fit is example of this approach. The cost of the algorithm is $O(1)$.

- TLSF (Two-Level Segregated Fit): TLSF is constant-time, good-fit allocator\textsuperscript{14}. Good-fit means it tries to allocate the block of available nearest higher size. It is based on combination of the segregated and bitmap fit mechanism. It contains two-dimensional array. The first dimension splits the free blocks in ranges of 2 i.e first level index $i$ splits blocks is size $[2^i, 2^{i+1}]$. The second dimension splits the first level range further. The number of such ranges is represented by the parameter $L$. Each entry in the two-dimensional array points to free blocks corresponds to that range. It maintains $\text{FL}_{\text{bitmap}}$ which indicates whether a free block is present in each row. Hence $\text{FL}_{\text{bitmap}}$ size indicates number of rows of the array where each bit indicates row. It also maintains the $\text{SL}_{\text{bitmap}}$ for each row which indicates the presence of free block in the second level range. Its length is decided by parameter $L$.

The figure 3.8 shows the instance of two-dimensional array of TLSF allocator. Taking the array shown in the figure 3.8 as instance of TLSF allocator and request for a block of size 74 is served by TLSF allocator as follows
• first level index (fl) is calculated using \( \log_2(r) \) and it is 6.

• Now it checks whether that level is free or not using FL\_bitmap if not it adjust fl to further level using FL\_bitmap.

• Second level index (sl) is calculated using \( r - 2^{fl}/2^{fl-L} \).

• Since sl calculated using above formula may contain list of size 72, 73 to avoid searching of such list it follows good fit rather than best fit and hence it increments sl. Hence sl is 2.

• Now it checks whether that level if free using SL\_bitmap. Since it is not it adjusts sl using SL\_bitmap to next free level and in this case it is 5.

• Now a block is removed from the list pointed by fl and sl, block is segregated and the remaining block is added to the position corresponding to its size and modify FL\_bitmap and SL\_bitmap accordingly i.e if there are no other free blocks set the bit as zero.
Chapter 4

Compression Algorithms

4.1 Introduction

As we have described in section 3.3 CRAMES select pages and compress them and store in the RAM itself so that effective RAM memory utilization increases. This method is called as Compressed RAM. The compression algorithms to be used for Compressed RAM should have following properties:\[1\][1]

- Compression should be lossless. There should be one-to-one mapping from compressed page to uncompressed page and vice versa. If the compression algorithms is lossy then it is not possible to bring back the uncompressed page from compressed page hence such type of algorithms cannot be used in Compressed RAM.

- Compression should be in memory. The compression algorithms should take less memory overhead.

- Compilers usually make alignment of data so that the processor can fetch the data easily. Example In C language the data elements of structures are word aligned. The compression algorithm should consider these type of optimization.

- Compression may be asymmetric. The time taken to compress the page may be different to uncompress the page.

- The metric used to evaluate the compression algorithms is compression ratio. Compression ratio is the ratio of size of compressed page to the size of the uncompressed page. The algorithm with good compression ratio, energy efficient and low memory overhead is desirable to be used in Compressed RAM.

Some compression algorithms are:\[1\]:

- WK in-memory compression family of algorithms developed by Paul wilson and scott F. Kaplan are designed keeping in view of 32 bit system.
• WK 4*4 is variant of WK compression which uses 4*4 set associative dictionary of recently seen words. They have highest compression ratio.

• WKdm compression family uses direct mapped dictionary of recently seen words, hence they are fast and gives good compression ratio.

• MiniLZO is the light weight subset of the LZO where compression is very fast and requires only 64KB of memory. Decompression is simple, fast and requires no extra space. LZO algorithm is thread safe and is lossless. LZO compression algorithms is 4-5 time faster than fastest zlib compression algorithm.

<table>
<thead>
<tr>
<th>Application</th>
<th>Algorithm</th>
<th>Characteristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>bzip2</td>
<td>Burrows-wheeler Transform</td>
<td>Compression ratio within 10 percentage of the state of art algorithms</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Relatively slow compression</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Decompression is faster than Compression</td>
</tr>
<tr>
<td>Zlib</td>
<td>LZ-family</td>
<td>Fast</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Compression/Decompression</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Good compression ratio</td>
</tr>
<tr>
<td>LZRW1-A</td>
<td>LZ-family</td>
<td>Very fast compression/decompression</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Good compression ratio</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Very low working memory requirement</td>
</tr>
<tr>
<td>LZO</td>
<td>LZ-family</td>
<td>Very fast compression</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Extremely fast decompression</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Favors speed over compressibility</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Low working memory requirement</td>
</tr>
<tr>
<td>RLE</td>
<td>Run-Length Encoding</td>
<td>Very simple and extreme fast</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Poor compression ratio for most data</td>
</tr>
<tr>
<td></td>
<td></td>
<td>No working memory requirement</td>
</tr>
</tbody>
</table>

Table 4.1: Comparison of various Compression algorithms[11]
4.2 Results

We need to compress the swappable pages and place it in compressed area. Hence compression algorithm is to be selected to accomplish the work. We have analyzed the performance of four well known compression algorithms namely LZO [13], BZIP2 [10], RLE, ZLIB [9]. The data on which we have to analyze the performance is main memory pages. We dumped the main memory pages to file using tools memdump, LiME(Linux Memory Extractor). We have conducted experiments and analyzed the algorithms across the following metrics.

- Average compression time
- Average decompression time
- Average Compression ratio
- Number of pages with compression ratio less than 1 (incompressible pages)
- Number of pages with compression ratio more than 2 (good compressible pages)

Experimental setup:-
- Operating System :- Ubuntu 12.04
- RAM :- 4GB
- Processor :- Intel i3 @ 3.30 GHZ
- Total Number of pages considered :-1023065 pages

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Average compression time(in secs)</th>
<th>Average decompression time (in secs)</th>
<th>Average compression ratio</th>
<th>percentage of pages with compression ratio &gt; 2</th>
<th>percentage of pages with compression ratio &lt;1</th>
</tr>
</thead>
<tbody>
<tr>
<td>BZIP2</td>
<td>0.000721652</td>
<td>0.00013264</td>
<td>3.059722272</td>
<td>876.9153475</td>
<td>3.6071022</td>
</tr>
<tr>
<td>LZO</td>
<td>0.0000102165</td>
<td>0.00000672605</td>
<td>2.565793356</td>
<td>66.60828</td>
<td>5.5760875</td>
</tr>
<tr>
<td>RLE</td>
<td>0.0000209695</td>
<td>0.000019247</td>
<td>1.709036531</td>
<td>37.9047275</td>
<td>9.1827987</td>
</tr>
<tr>
<td>ZLIB</td>
<td>0.000156692</td>
<td>0.0000254121</td>
<td>3.292300297</td>
<td>79.7678544</td>
<td>3.3061438</td>
</tr>
</tbody>
</table>

4.2.1 Compression Ratio

Compression ratio is the ratio of size of data before compression to size of data after compression. So if the compression ratio of a page is lesser then 1 it indicates that page is incompressible. Higher the compression ratio the lesser the size of data after compression.
Figure 4.1 indicates the average compression ratio of four compression algorithms. Zlib have highest compression ratio of 3.292300297 than others. RLE have less compression ratio of 1.709036531 compared to others.

4.2.2 Decompression Time

Figure 4.2 shows the average time taken by each compression algorithm to decompress the page. LZ4 takes less time of 0.00000672605 seconds than others. BZLIP2 consumes more time of 0.00013264 seconds compared to others.

4.2.3 Compression Time

Figure 4.3 indicates the average time taken by each compression algorithm to compress single page, two consecutive pages, three consecutive pages at a time. LZ4 consumes less time of 0.0000102165 seconds when compressed single page. BZIP2 takes more time of 0.000721652 seconds when compressed single page.
4.2.4 Good Compressible Pages

Pages with compression ratio > 2 i.e whose size after compression is reduced more than by half are said to be good compressible pages. Figure 4.4 shows the numbers of good compressible pages out of 1023065 pages tested for four compression algorithms. Zlib have highest number of good compressible pages of 816077, whereas RLE have least number of 387790.

4.2.5 Incompressible Pages

Pages with compression ratio <1 i.e whose size after compression increases are said to be incompressible pages. Figure 4.5 represents the number of incompressible pages out of 1023065 pages tested for four compression algorithms. Zlib have less number of such pages counted to 33824, whereas RLE have highest number of 93946 such pages.
4.2.6 Statistics of Zlib

Figure 4.6 indicates the percentage of pages with inverse of compression-ratio in the range using Zlib.

4.2.7 Conclusions

- One can observe from figure 4.2 and figure 4.3 decompression time and compression time are asymmetric. Time taken to decompress is less than compression time.

- From figure 4.3 we can observe the time to compress in not propositional to size of the block. So compressing a block takes less time than compressing the same block half each time. So we can compress two pages together so that the time taken to compression is reduced when they need to be placed in compressed area at low memory situations. But it will restrict us to decompress both the pages when one of them is accessed which incurs more overhead.
• There are almost 3-5 percentage of pages whose size increases after compression.

• There are almost 10 percentage of page whose size is 75 percentage after compression even in best performed ZLIB algorithm. Selecting a page from such a list will not help us to save memory.

• The LZO and ZLIB performs well. ZLIB performs better than LZO. But the compression and decompression time is more than LZO. Whereas RLE performance very bad. Its a tradeoff to select among LZO and ZLIB. ZLIB performance better than LZO at the cost of more compression/decompression time.
Chapter 5

Design of Compressed RAM

At low memory situations to allow execution of process usually the operating system maintains swap space on secondary device and swap the pages to this area so that we get some free frames in main memory. But Android does not have swap space because of the following reason:\[15\]:

- Memory is scarce and adding more memory increases device costs
- Flash storage suffers from wear-leveling issues hence the using it as swap device decreases the life span of the flash memory

When there is Scarcity of the memory Android Low memory killer kills some of the process to free the memory. Objective of the Compressed RAM project is to use part of the RAM as swap device. The pages swapped to the device are compressed and stored in the RAM itself. Hence the effective memory of the device increases. Compressed RAM is a technique which compress the pages in the RAM and stores back in the RAM itself. Hence the effective data to store in the RAM increases. It helps to get rid of killing applications at low memory situations to some extent which is the case in Android. Compressed RAM keeps apart some part of the RAM as compress memory and it stores all the compresses pages in this compress memory. Compressed RAM techniques CRAMES, Compocache depends on Linux swapping mechanism for selecting the pages to compress and store in compressed area \[11\] \[15\]. As described in section 1.3 Linux swapping mechanism inturn depends on LRU based mechanism \[8\]. The Compression Cache technique which is variant of Compressed RAM depends on LRU mechanism to select pages to be compressed\[8\].

5.1 CRAMES Approach

Compressed RAM for Embedded System(CRAMES) is the software based compression technique\[11\]. It compress data part. It increases effective memory without increasing physical memory. Here main memory is divided into two parts one stores the uncompressed
data and other stores the compressed data.

CRAMES works as follows:-

- Set part of the RAM as compressed area.
- The kernel selects some of the pages to write to this compress area usually at low memory area.
- The pages which are written to this area are compressed and stored.
- The selection of pages to be compressed is depended on Linux swapping mechanism\textsuperscript{11} which follows LRU based approach\textsuperscript{5}.
- When a process want to access the page which is stored in the compressed area it decompress the page and stored in uncompressed area of the RAM.

5.1.1 Observations

The selection of pages to be compressed is depended on Linux swapping mechanism which follows LRU based approach. When a page is selected they compress it and place it in compressed area. Figure 4.5 indicates there are almost 3-5 percentage of pages whose size increases after compression. Figure 4.6 indicates there are almost 10 percentage of page whose size is 75 percentage after compression even in best performed Zlib algorithm. Selecting a page from such a list will not help us to save memory. In present approach they place the compressed page in compressed area irrespective of the size after compression which leads to no saving of memory and wasteage of CPU cycles when selected page is incompressible.

5.2 Proposed System

In our approach we will select pages not only based on access time (least recently used) but also we consider compression ratio. This section describes about the new approach.

5.2.1 Pages that should be targeted

Table 1.1 describes the classification of pages. Unreclaimable pages are kernel pages which are frequently accessed and hence selecting them as target to compress will lead to bad performance. Discarded pages are unused pages and they are claimed by OS and are allocated to other process. Hence they are not our target. Syncable pages are pages backed by file and they are written back to file at low memory situations. Hence even they are not our target. Swappable pages are user pages which are our target.
5.2.2 Properties of pages

The following properties will help us to decide which page to be selected to compress it at low memory situations.

Access Time

We have to keep track of access time of each page. This attribute will determines how long back the page is accessed. The longer the page is accessed the better to select this page because it is more unlikely to be accessed in near future.

Compression Ratio

The selected page for compression may not be compression friendly i.e the size after compression of page when compared to page size may not have much difference or even data size may increase after compression. Selecting such a page will not help us in saving the memory at the cost of CPU overhead for compression and decompression. But the compression ratio of particular page is found only after compression it. We will try to keep track of that information after compression such that it will not likely be the target for next time.

One way to handle this is to maintain binary bit for each page which indicates whether the page is compressible (Compressing the page will saves us memory) or not. Initially all the pages are compressible(bit is 1) only after it is selected as target and compressed we will come to know whether it is compressible or not and will update the bit accordingly. Another more good way is keeping track of not only compressible/incompressible but how good is it compressible. To do this we will keep an variable which indicates how good the page is compressible. Initially all the pages are compressible. Once it is selected as target we will update the variable and it helps us to consider this attribute hereafter when selecting this page.

Based on above two properties we will calculate the score of the page. We will maintain two list, list with pages whose compression ratio is not known and list of pages whose compression ratio is known. Whenever a page is accessed the page is updated in the list based on new score. When a page in the list of pages whose compression ratio is known is modified we will move page to other list as compression ratio is no more valid as content of the page is changed. We will select the pages from both of the list at low memory situations to compress them. The second list is sorted base on score which we followed is \(\frac{access\ time}{10} + \frac{100}{compression\ ratio} \times 5\)
As score calculated depends even on compression ratio it will not favors the pages whose size in not reduced much after compression for pages whose compression ratio is known.
The Enhanced Compressed RAM approach is explained below:-

- Set part of a RAM as compressed area
- Initially we will maintain an LRU list of pages in RAM
- Whenever a page is allocated to process in RAM, an entry is made in LRU list
- At low memory situations we will find process with highest number of pages in RAM and pages of that process from LRU list are selected, compress them and store them in compressed area if and only if its size reduced to considerable level and its compression ratio attribute is updated until we reach high memory threshold. If the size of page after compression does not reduce to considerable level placing them in compressed area does not save us memory and hence it is kept in the RAM itself and an entry is registered in second list which is sorted based on compression ratio and access time, as described above.
- Whenever a page in RAM is accessed its position in list where it exists is updated.
- Whenever a page in compressed area is read accessed it is uncompressed and placed it in the RAM and an entry is made in second list as its compression ratio attribute is known.
- Whenever a page in compressed area is write accessed it is uncompressed and placed it in RAM and an entry is made in LRU list as the compression ratio is no more valid as the content is changed.
- Here after at low memory situations pages of process are selected from both the lists, where one list is constructed based on access time and other is constructed based on access time and compression ratio based on formula described above. Algorithm 1 given below explains the same
Algorithm 1 Enhanced Compressed RAM Page Selection

1: List lru; //least recently pages list
2: List sec; //List of pages whose compression ratio is known
3: function PAGEACCESSHANDLER
4:     if read access then
5:         if Page in Uncompressed Area then
6:             Update its position in lru;
7:         else if Page in Compressed Area then
8:             Uncompress it, place it in compressed area and add it to sec;
9:         else
10:             handle pagefault;
11:     end if
12: else
13:     if Page in Uncompressed Area then
14:         Update its position in lru;
15:     else if Page in Compressed Area then
16:         Uncompress it, place it in compressed area and add it to lru as the compression ratio is no more valid;
17:     else
18:         handle pagefault;
19:     end if
20: end if
21: end function
22: function LOWMEMORYHANDLER( )
23:     pid= process with highest number of pages in RAM
24:     while low memory situation do
25:         //divide least recently used page of process pid with 10 and add 10
26:         //10 is added as most of the pages( 80 % ) will compress to 75 % size
27:         score1 = lru.tailelementofprocess(pid).accesstime/10 + 10;
28:         ele = sec.tailelementofprocess(pid);
29:         score2 = ele.accesstime/10 + 100/(ele.compressionratio * 5)
30:         if score1 < score2 then
31:             select page from lru and compress it;
32:             if size after compression > 75 % of page size then
33:                 remove it from lru, set its compression ratio;
34:                 add it to sec;
35:             else
36:                 remove it from lru, allocate memory in compressed area;
37:                 free the Frame;
38:             end if
39:         else
40:             select page from sec, remove it from sec, allocate memory in compressed area;
41:             free the Frame;
42:         end if
43:     end while
44: end function
1: Set part of the RAM for compressed area
2: Maintain two lists lru list of pages whose compression is not known, sec list whose compression ratio is known
3: Initially as compression ratio of any page is not known, every page in RAM have entry in lru list
4: At low memory situations call lowMemoryHandler()
5: when a page is accessed call pageAccessHandler()

5.2.3 Results

We have simulated the both the approaches where selection of pages to be compressed in based on access time in first approach and other is based on access time and compression ratio. The events to access the pages are generated at random time interval.

Simulation setup:-

- Number of pages in RAM are 1000
- Number of pages preserved as compressed area taken from RAM are 100
- Low memory threshold is 100 pages
- High memory threshold is 200 pages
- Compression algorithm used Zlib
- Scheduling algorithm followed Roundrobin
- Time quantum for each process 1 milli seconds
- Number of pages for each process are allocated randomly in the range 100 to 200
- Page access events are randomly generated
- Events are placed in event queue according the event time
- Read page miss processing time 100 micro seconds
- Read page hit processing time 1000 nano seconds
- Write page miss processing time 100 micro seconds
- Write page hit processing time 1000 nano seconds
- Time taken to compress 100 nano seconds
- Time taken to decompress 100 nano seconds
Memory utilization

Figure 5.1 shows the free memory with respective to time line in traditional CRAMES approach and modified new approach. Number of process executing are 20. At time 66 milli seconds the system faced low memory situation and the compressing of pages started. We can see the free memory in the later case is more than other. In first case the system has exhausted the memory and out of memory killer is called at time 91 milli seconds. Where as in later case it happened at 120 milli seconds.

![Figure 5.1: Free memory with respective to time line](image)

Memory saving

Figure 5.2 indicates the memory saved in both the cases. In traditional CRAMES approach we are able to store 171 pages in compressed area when the system exhausted the memory. Where as in later case we are able to store 399 pages in compressed area. we almost doubled the memory saving.

Overhead

Figure 5.3 indicates the overhead of the modified approach against traditional approach. Figure 5.3 shows the number of compressions and number of decompression in both the case. Even though the overhead in second case increases but it because of incompressible pages.
Figure 5.2: Number of pages stored in compressed area

Figure 5.3: Number of compressions and decompression
Chapter 6

Conclusions

Modern devices with no swap space results in killing of process at low memory situations which was the case in Android. Use of 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 there is need of new techniques like sub paging to increase the efficiency. Compression algorithms to be used in Compressed RAM should have good compression ratio, less working memory and less compression/decompression time. Results shows Zlib and LZO performs well and its trade off to choose between them. As there are more than 10 % of pages whose size is more than 75 % of page size after compression even in best performed Zlib, depending on Linux kernel swapping for Compressed RAM will not give good performance. It is meant in situations where compression is not involved and redesign of swapping system in devices where Compressed RAM is used as swap space is needed and the results shows the increase in performance.
Chapter 7

Further work

Implementing the proposed system on Android device. Evaluating the results in such a device. Since in the case of Android devices most of the time the device is idle where we can compress some of the least recently used pages in advance for the sake of knowing the compression ratio information as they are the one likely to be swapped in near future which causes them to be compressed. We have invalidated the compression ratio even if bit of page content is changed, in such cases it does not changes the compression ratio much, it may increase or decrease little. Assuming the same compression ratio because it is more likely to be similar to that of before, checking the performance of the system under such assumption. Design system with above assumption and finding the compression ratio of such dirty pages when the system is idle.
Acknowledgements

I would like to thank Dr. D.B.Phatak for his tremendous support and help. I would also like to thank Nagesh, Rajesh, Bala Murali Krishna, Shashank for their inputs.
References


