Design and Implementation of CompressedCache

M. Tech. Project Stage-1 Report

Submitted in partial fulfillment of the requirements
for the degree of

Master of Technology

by

Subbanjaneyulu Reddy
Roll No: 113050021

under the guidance of

Professor D.B. Phatak

Department of Computer Science and Engineering
Indian Institute of Technology, Bombay
Mumbai
# Contents

1 **Introduction**
   1.1 Memory Management in OS ..................................................... 1
      1.1.1 Address space .......................................................... 1
      1.1.2 contiguous memory allocation ........................................ 1
      1.1.3 Segmentation ........................................................... 3
      1.1.4 Paging ................................................................. 3
      1.1.5 Translation Look Ahead Buffer ........................................ 4
      1.1.6 Page Replacement ..................................................... 4
      1.1.7 Linux Page Frame Reclaiming Algorithm ............................ 6

2 **Memory Management in Android** ............................................. 8
   2.1 Memory Management ........................................................ 8
      2.1.1 Dalvik Virtual Machine .............................................. 8
      2.1.2 Application ........................................................... 8
      2.1.3 Categories of Process ............................................. 8
   2.2 Activity LifeCycle .......................................................... 9
   2.3 States of Activity .......................................................... 9
      2.3.1 Active ................................................................. 9
      2.3.2 Paused ............................................................... 10
      2.3.3 Stopped ............................................................. 10

3 **Literature Survey** ............................................................. 12
   3.1 Swap compression .......................................................... 12
      3.1.1 Read/Write path ..................................................... 13
      3.1.2 Swapping In .......................................................... 14
      3.1.3 Swapping Out ....................................................... 14
      3.1.4 Cache-cleaning algorithm ........................................ 14
   3.2 Virtual memory in system with Flash memory as secondary storage .... 14
      3.2.1 Flash Memory ....................................................... 15
      3.2.2 Subpaging ........................................................... 15
      3.2.3 Hotcache ........................................................... 15
   3.3 Online Memory compression for Embedded systems ....................... 16
      3.3.1 Code Compression ................................................... 17
      3.3.2 Data Compression .................................................. 17
      3.3.3 CRAMES .............................................................. 17
      3.3.4 Virtual Memory Swapping ......................................... 17
      3.3.5 Request Handling ................................................ 18
   3.4 Compression Algorithms .................................................. 18
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Main memory shared among process</td>
<td>1</td>
</tr>
<tr>
<td>1.2</td>
<td>Address translation From virtual to physical address</td>
<td>2</td>
</tr>
<tr>
<td>1.3</td>
<td>contiguous allocation of memory to process</td>
<td>2</td>
</tr>
<tr>
<td>1.4</td>
<td>Memory allocation to process using segmentation</td>
<td>3</td>
</tr>
<tr>
<td>1.5</td>
<td>Address Translation using paging</td>
<td>4</td>
</tr>
<tr>
<td>1.6</td>
<td>System calls invoked when the pages are reclaimed in linux</td>
<td>6</td>
</tr>
<tr>
<td>1.7</td>
<td>Transitions among the two LRU lists</td>
<td>7</td>
</tr>
<tr>
<td>2.1</td>
<td>State of Process</td>
<td>9</td>
</tr>
<tr>
<td>2.2</td>
<td>Activity Lifecycle</td>
<td>10</td>
</tr>
<tr>
<td>3.1</td>
<td>Conceptual vision of the compression and cache mechanism</td>
<td>13</td>
</tr>
<tr>
<td>3.2</td>
<td>Graphical example of the two kinds of read hits</td>
<td>14</td>
</tr>
<tr>
<td>3.3</td>
<td>Compression and Cache mechanism with two different swappig paths</td>
<td>15</td>
</tr>
<tr>
<td>3.4</td>
<td>Time-Frequency breaks the intra page locality</td>
<td>16</td>
</tr>
<tr>
<td>3.5</td>
<td>Mechanism of Two level LRU</td>
<td>17</td>
</tr>
<tr>
<td>3.6</td>
<td>Logical structure of swapping mechanism on compressed RAM device</td>
<td>18</td>
</tr>
<tr>
<td>3.7</td>
<td>Handling request in CRAMES</td>
<td>19</td>
</tr>
</tbody>
</table>
Acknowledgements

I would like to thank Dr. D.B. Phatak for his tremendous support and help. I would also like to thank sir Nagesh and Rajesh for their inputs.
Abstract

Android is a software platform and operating system for mobile devices, based on the Linux kernel. It was developed by Google and later the Open Handset Alliance. Memory is a scarce resource in embedded system like Android. Increasing memory often increases packaging costs, cooling costs, size, complexities and power consumption. The complexities and resource demands of modern embedded systems such as Android smart phones are constantly increasing. In order to support applications such as games, the memory requirements of embedded systems have grown at a much faster rate than was originally anticipated by their designers. The report go through the memory management in conventional OS like linux, Android and some techniques that increases the performance of the system when flash memory as secondary storage, use of cache for swap space, use of RAM as swap in swapless device.
Chapter 1

Introduction

1.1 Memory Management in OS

1.1.1 Address space

The memory space a process can use as it runs is called as Address space[13][7]. Each process can have address space as long as the space generated by the virtual address. The abstraction provided by the address space allows each process to have the same numerical values but it is logically distinct. Virtual memory allows the process to have address space more than that of the physical memory. This allows the machine to run the process which have more address space than the physical memory. Due to introduction of multi programming there is a need of memory protection as one process can not use the memory of some other process.

```
fffff (high memory)
  .  operating system
  .
  80000
  7ffff
  .  user process
  .
  00000 (low memory)
```

Figure 1.1: Main memory shared among process

Since the logical address is different from physical address. The process views logical address but actual the process reside in physical address and hence there is a need of dynamic address translation.

1.1.2 contiguous memory allocation

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

Translation

let the base address of the process where the allocation starts be X. So the any address logical address Y of the process is at X+Y in main memory.
Protection
when ever we need to access logical address Y we need to compute X+Y and see if it is beyond the size of the address space of the process if so its illegal to access it and hence do not allow it. This way the OS can protect the process from accessing other process memory.

Advantages
- simple in implementation, just need 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 fragmentation. Where the unused space in main memory is distributed across the memory and can not be used to allocate to the process because we need it to be contiguous.
1.1.3 Segmentation

The logical address of the process is divided into segments like code segment, data segment, etc. Each segment is allocated continuously in the physical memory\(^3\). Virtual address is divided into virtual segment, offset.

![Figure 1.4: Memory allocation to process using segmentation](image)

Protection and Translation

For each segment we keep track of the starting and ending physical address. When ever process need access to an logical address x. OS computes which segment the x falls in and compute the physical address based on starting address of the segment and check against ending physical address.

**Advantages**

- Reduces the external fragmentation.
- Works well for sparse address space.

**Disadvantages**

- The process cannot use all the virtual address because the segment size are of different size.
- Still it suffers from external fragmentation.

1.1.4 Paging

Logical Address is divided into equal size chunks called as pages. Physical address is divided into equal size chunks called as frames or page frames. Any page of any process can fit into any frame because both are of same size. We keep track of which page goes into which frame by using an table called as page table\(^3\). Each process has its own page table. Virtual address is divided into two parts, page number and page offset.
Translation

When ever process generates an virtual address, OS obtain 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. Else an invalid bit is set hence page fault occurs 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 leads to physical address.

Protection

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

Advantages

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

Disadvantages

- Still there is internal fragmentation in the last page of the frame.
- 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. So TLB cache stores the page number to frame number mapping[3] . So when a hit in TLB occurs only one main memory access and TLB access is required. TLB access is less expensive.

1.1.6 Page Replacement

When an page fault occurs and there is no free frame 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] [11].
Global replacement algorithm

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

Local replacement algorithm

When a page is to be replaced the page from 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.

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.

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.

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

When ever an page is referred update it 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.

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 will be used which stores the slot in the circular buffer which has to be next examined. When ever a page is referred the reference bit is set as 1. When a page fault occurs the slot which is next to be examined 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 a infinite cycle when all slots has reference bit as 1.

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
1.1.7 Linux Page Frame Reclaiming Algorithm

The Linux Page Frame Reclaiming Algorithm (LFRA)\cite{6} has to make sure that there is some free memory. Pages are classified into four classes and the following table shows them.

<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, reserved paged</td>
<td>Impossible</td>
</tr>
<tr>
<td>Swappable</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\cite{6}

Mapped page is the one which is backed by the file. It corresponds to a particular block in the filesystem. Where as anonymous page does not correspond to any file it is part of process space like stack, heap etc.

The reclaim memory 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 page table changes to be done. When a shared page is reclaimed the page table of all the processes 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 same Each Process maintains

Figure 1.6: System calls invoked when the pages are reclaimed in Linux \cite{6}
two LRU, 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 list are reclaimed.

![Figure 1.7: Transitions among the two LRU lists](image)

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 only one process at a time holds the swap token. Pages belonging to the swap token owner are never recaimed. kswapd is the kernel thread which swpas the swap pages(anonymous pages).
Chapter 2

Memory Management in Android

2.1 Memory Management

Android is Linux-based OS. All the basic OS operations like I/O, memory management, and so on, 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[9], giving all responsibility for memory and process management to the Android runtime, which stops and kills processes as necessary to manage resources.

Dalvik is a register-based virtual machine that’s 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 footprint. 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 at least as high as priority as the 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[9]: 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 to 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 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. Because Services dont interact directly with the user, they receive a slightly lower priority than visible Activities. They are still considered to be foreground processes and wont 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 when ever resources are less. Android uses last seen first kill pattern to kill the process[9].

- 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.

### 2.2 Activity Lifecycle

An activity is a single, focused thing that the user can do. All the activities interact with the user. Activity Stack[5] is used by Android to maintain the activities. When a 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 by 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[8]

When an activity is paused or stopped and at low memory situations it is killed by android. When they are displayed to the user it must be completely restarted and resorted to previous state.

![Activity Lifecycle](image)

Figure 2.2: Activity Lifecycle

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 in onCreate(), and release all remaining resources in onDestroy(). For example, if it has a thread running in the background to download data from the network, it may create that thread in onCreate() and then stop the thread in onDestroy().
The visible lifetime of an activity happens between a call to `onStart()` until a 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()` until a 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.
Chapter 3

Literature Survey

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 when 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 the multiple of size of the a cache buffer.
- Swap out[4]: When ever the system need an free page and all the page are in use then the system send some pages to swapping 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[1]: The page of the file are called as page cache pages. They are also called as file backed pages
- Swap-cache: The pages that are not file backed are called as swap cache pages. They are also called as Anonymous pages.

When even an page is send to page to the disk, it is compressed and places 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

When ever a read hit is found it may be 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 send 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 not in the cache because it is send recently swapped out, but it is in the cache because it is brought along with 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 given page is requested, the system prefetchs all the pages in the same disk block and if any of these pages is requested later on, a read hit due to read occurs.

- 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 Show that the Hits due to read are very less. The reason is the order in which we swap out the page 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 that 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 and decompress the page and send to swapping module. The block is not places 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 compresses 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\[4]. 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.

3.2 Virtual memory in system with Flash memory as secondary storage

Operating system implements virtual memory system to allow process with more than physical address 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. The conventional Virtual memory system degrades the system performance and decreases the life of secondary storage when Flash memory is used as secondary storage.

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

\[4\]
3.2.1 Flash Memory

Flash memory works little different from magnetic disc. The operations allowed on Flash memory are read, write, erase. Read operation is same as the read in magnetic discs. Update operation is different from that of magnetic discs. When ever we want to update a page on the flash memory we need to find an free page and write on to it the 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 the 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 Subpaging

In conventional virtual memory system when ever a dirty page is to be written back to secondary storage its write entire page. So when the secondary storage is flash memory it results in 8 writes. 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 in 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 size of the 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 into 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 Hotcache

To reduce writes to flash memory frequent writes are kept to SRAM. Hotcache is organized as a fully associative cache. The block size of Hotcache is page of the flash memory. The cache management is done by using one of the three policies.

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 gaurantee allocation as in case a.
Time-Frequency: Each write request is cached in Hotcache. The replacement is based on 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 in to flash 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 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 to be 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.

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
decompress 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 decompressed 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, the code usually 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 small portion of memory and large 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 no 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 decompresse and given to the precess 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 that can fit the size of the compressed page is taken and is allocated for compressed page and finally the old page is freed.

### 3.4 Compression Algorithms

The compression algorithms to used for Compressed Cache should have following properties[8][1]:

- compression should be lossless. There should be one-one mapping from compressed page to uncompressed page and vice versa. If the compression algorithms is lossy then we cannot bring back the uncompressed page from compressed page hence such type of algorithms cannot be used in Compressed cache.
- Compression should be in memory. The compression algorithms should take less memory overhead
Figure 3.7: Handling request in CRAMES

[8]
• 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 Cache.

Some compression algorithms are[8][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. Due to this assumption it is not possible to use this family of algorithms for 64 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 ration.

• 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 and fast and requires no extra space. 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>compress ratio within 10 percentage of the state of art algorithms relatively slow compression Decompression is faster than Compression</td>
</tr>
<tr>
<td>zlib</td>
<td>LZ-family</td>
<td>Fast Compression/Decompression Good compression ratio</td>
</tr>
<tr>
<td>LZRW1-A</td>
<td>LZ-family</td>
<td>Vert fast compression/decompression good compression ratio very low working memory requirement</td>
</tr>
<tr>
<td>LZO</td>
<td>LZ-family</td>
<td>very fast compression extremely fast decompression favours speed over compressibility low working memory requirement</td>
</tr>
<tr>
<td>RLE</td>
<td>Run-Length Encoding</td>
<td>Very simple and extreme fast poorer compression ratio for most data no working memory requirement</td>
</tr>
</tbody>
</table>

Table 3.1: Comparision of various Compression algorithms[8]
Chapter 4

Problem Statement

Android does not have swap space because of the following reason[2]:

- 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 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.

- Design and Implement Compressed Cache for AKASH tablet.
- Performance analysis with CompCache and without CompCache
- Performance analysis by using different compression algorithms
- Dyanamically changing the part of the RAM which is used as swap device.
- Use both part of RAM and flash memory as swap device with concept of subpaging and analyse the performance.

Design:

- Part of the RAM say 10 percentage is fixed for swapping purpose.
- When ever there is scarcity of memory some of the pages is choosen as pages to be swapped based on some replacement algorithm.
- The selected page is compressed and as the compressed page size may vary we use some allocation algorithm like best fit, next fit etc to allocate room for the compressed page in the swap area of the RAM.
- When ever a compressed page is need by the process the system finds it in swap area,allocates frame for it, decompress the page and place it in allocated frame
- Performance is done by testing the memory utilization and cpu utilization in various work loads environment.
- Swap area should not be fixed as it may wastes the memory. So we may change the allocator to grant requested memory for the swap device and design the device to request the required memory as on when needed
Chapter 5

Conclusions

Modern devices with no swap space results in killing of process at low memory situations. 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 extent at low memory situations. Modern embedded system where flash is used as secondary storage will suffer from performance when conventional virtual memory system is used. Hence there is need of new techniques like subpaging to increase the efficiency. Use of both RAM and flash memory as swap space will increase both effective memory and performance of the system.
Bibliography


