A Memory Management Scheme for Enhancing Performance of Applications on Android

Kumar Vimal
Department of Information Technology
ABV - Indian Institute of
Information Technology and Management
Gwalior - 474015, India
Email: kumarvimal1992@gmail.com

Aditya Trivedi
Department of Information Technology
ABV - Indian Institute of
Information Technology and Management
Gwalior - 474015, India
Email: atrivedi@iiitm.ac.in

***Abstract*—Android OS is an unbridled success in the mobile market. Memory management has always been an area of concern to developers of large applications and also to consumers who want a seamless user experience. Memory leaks in applications have been talked about a lot in the past where subtle vulnerabilities often result in unwanted consequences such as application crashes. Memory is a limited resource for any device especially in portables (mobiles, tablets) and effective memory management is of utmost importance in determining the responsiveness of any system. Android contains modified Linux kernel and the task of managing memory is done by Activity Manager Service (AMS) and Low Memory Killer (LMK). Android features a least recently used reclamation process and adjustment based low memory handling. But this scheme may kill processes that may be required in a short interval of time leading to more number of memory loading cycles taking time in the order of 3 - 5 seconds. Previous research done in this area has attained 22.1% loading latency improvement of applications and this research is an endeavor to do better by using more appropriate cache management techniques. The main goal of this research is to design a new memory management scheme for the Android operating system that takes into account application usage patterns of the user to decide the applications that have to be killed from the main memory and dynamically set the background process cache limit based on hit rate and number of applications of user's interest.**

Keywords*—memory management, LRU, Activity Manager Service, Low Memory Killer, cache management, application usage patterns, Android.

I. Introduction

An embedded operating system is a type of operating system that is embedded and specifically configured for a certain hardware configuration. Hardware that uses embedded operating systems is designed to be lightweight and compact, forsaking many other functions found in non-embedded computer systems in exchange for efficiency at resource usage. This means that they are made to do specific tasks and do them efficiently. Embedded operating systems have small memory footprint and suffer with power and processing constraints. Examples- Symbian, iOS, Windows Phone, Android, blackberry OS, Firefox OS, etc. Android is the most popular mobile operating system used by many mobile devices such as smartphones, tablets and portable PCs supporting many RISC (reduced instruction set) architectures like ARM, MIPS and X86. Linux kernel version 2.6 (monolithic kernel) is modified for usage in Android with approximately 115 patches. This provides basic system functionality like process management, memory management, device management like camera, keypad, display etc.

978-1-4673-6670-0/15/$31.00

A. Activity Manager Service (AMS)

AMS is a service that executes at the framework layer on Android and is responsible for receiving and responding to user requests in an appropriate way. In Android during boot time [1], all services are initiated by the zygote system process (AMS, window manager). After the start of AMS, other system applications are loaded like User Interface, Home Window, contacts, keyboard and other third party applications with RECEIVE BOOT COMPLETED [2] permission. Functions of the Activity Manager include starting and killing processes and updating OOM Adj values of applications. Applications are dynamically ranked into importance categories, called adjustments, by the Activity Manager as listed below:

1) Foreground (active)-currently focused application of the user.
2) Visible process-an application process bounded to foreground process.
3) Service Process-a process running in the background.
4) Hidden Process-a disabled process not visible to the user but present on the device.
5) Content Provider-an application that provides structural data like calendar, contacts.
6) Empty processes are cached background processes that are already terminated but still present in main memory for fast loading on next usage.

B. Cached Background Process

When an app that was running on the screen is left, it stops running, but Android keeps its process in memory. This means that next time we want that app in the foreground, or next time it runs a service in the background (e.g. to check for email), the app can run again without Android having to load it from storage again. This means it starts faster and uses less battery. Apps that have been kept alive but not running in this way are called cached background processes. They still use some RAM, but Android will automatically remove them from RAM if it needs to free it up for running apps, so they don't affect the amount of RAM available to other apps.

C. Related Work

Researchers in [3] have proposed a complex Markov decision model for reclamation of memory on Android. They periodically inspect the stop queue to calculate the survival probability of app in the next inspection and those having low probabilities are killed based on a threshold. This paper has led to 22.1% better loading efficiency than the default LRU scheme. The study given in [4] is very relevant to present day scenario of phones having large RAM sizes where cache management is important to keep relevant processes in memory. Caching efficiency is optimized in this research paper.

The study done till now does not consider features like free memory, cached memory, duration of application use, last accessed time, temporal probability, etc. to get the importance of an app for deciding the process that has to killed in dynamic setting of background cache limit. This paper is an effort to include all these features to get better response time on average on the smartphone. In the support of the idea [5] shows that users install a large number of applications on their smartphone but only a subset of it is used with high frequency and [6] shows effective utilization of usage pattern of applications by user to optimize the user interface on the phone.

Section II gives the understanding of memory management on smartphone and problems in the existing scheme. Section III describes about the methodology and implementation of the proposed scheme. Section IV contains the results of the research work and conclusions are presented in section V.

II. Memory Management on Android Platform

Android OS uses a lot of techniques to solve memory crisis scenario like page reclamation, swapping, cgroups and Low Memory Killer (LMK) but mobile manufacturers don't use swap technology for commercial manufacturing because of performance issues related to writing on the flash storage again and again. The basis of this study is to look into the page reclamation technique that Android natively uses to free memory, find the problems associated with the scheme and develop an approach to manage memory in a more efficient manner.

A. Android Memory Analysis

Android consists of many components in different layers. Every component has different memory footprint. Its important to know how much memory will be required for every component. DDMS (Dalvik debug monitor server) [7] is an open source tool that helps in analysis of memory footprint of the different components.

B. Problems in AMS

AMS kills applications before Low Memory Killer can activate itself in case of low memory scenarios. This killing done by AMS does not take into account the frequency of access by user and is done on the basis of last accessed application only. The following problems exist in the AMS with regard to killing of applications.

1) Free memory size consideration: Number of empty applications and hidden applications should not be allowed to exceed a certain static threshold and should not be too less either as this decreases the overall performance of the system.

2) Frequency of application access consideration: Usage statistics of every user is different and it is important to learn the access pattern to make correct prediction of the sequence in which the applications must be killed based on parameters described in later sections. If such user interested applications can be protected from killing and are only killed in severe memory pressure then the number of load operations over a period of time can be improved drastically.

C. Dalvik Heap Limit and Global Memory

The only memory management that impacts activity lifecycle is the global memory across all processes. When Android decides that it is running low on memory, it needs to kill background processes to get some back. If an application is sitting in the foreground starting more and more activities, it is never going into the background, so it will always hit its local process memory limit before the system ever comes close to killing its process. An activity going into background is different from a process going into background. By default, all activities in an Android Application run in the same process, as explained on the Android Developer page. The global memory is basically the RAM. One can see this at the bottom of the screen when you go to the device Settings -> Apps -> Running. On the other hand, each process in Android has a separate memory limit, which is usually referred to as the Dalvik heap limit. It is described in the Android Developer Documentation as well [8]. The Dalvik heap limit is different on every device. In general, the newer the OS and the better the hardware, the bigger it usually is. It could be anywhere between 16 MB to 128 MB, maybe even more.

III. Proposed Algorithm and its Implementation

In modern smartphones the amount of RAM that is being put in is large compared to the previous generation. RAM sizes of 2GB and beyond is very common nowadays. So the previous work done on optimizing the memory reclamation process doesn't serve the purpose of bringing responsiveness to the loading of applications as the memory never goes too low to start the reclamation process by the AMS/ LMK. In these lines a better cache management scheme is required that keeps an optimum number of background processes and at the same time maintain less number of file operations that might be caused due to excessive page faults. The tradeoff of interest is the page cache memory and application cache memory competing against each other.

A. Hit rate vs Page Faults

Hit rate is defined as the number of times an application was brought into running state from cached state and is taken from a certain number of pre-decided slots to be observed. The hit rate increases when the number of background processes increase and high hit rate can be easily achieved by caching majority of the applications commonly used by the user. Page Faults on the other hand are the situations where an application has to loaded has to be loaded from the internal storage/SD card to the main memory in order to execute to completion. The amount of page cache in the memory can be increased by decreasing the number of background processes cached in the main memory. This tradeoff between the two limits calls for an algorithm that sets the background process limit

dynamically and caches the apps important to the user at the present scenario. This will help in minimizing page faults without sacrificing hit rates to a large extent.

B. Parsing of Memory Logs

Memory information is maintained by Android and it can be queried by a system app to allow reading of its constituents. We are mainly interested in 2 types of processes. We extract the running processes that are non-native and non-persistent in nature (those processes that lie in the user domain and not in the system domain) and cached Processes that are present in the logs. By polling the cached and running applications after fixed intervals of time, we can find out the recently launched applications and whether these newly added applications were revived from the cached zone or not.

C. Java Reflection API

Internal classes in Android that are not available to be used by an Android developer can be used by finding the dex file from where we want to extract the target class. This can be very helpful in setting the background process limit by using the ActivityManagerNative and IActivityManager classes to solve the purpose [9].

D. Timer Task and Polling Service

A timer is created and tasks are scheduled at fixed rate with a delay of 2 seconds between the start of each task. The polling is started with the start of the background service that is implemented along with the activity. The purpose of the task is to parse the memory logs, calculate the latest hit rate and query for the free memory available to the user. A decision is taken whether to increase or decrease the process limit based on the collected information.

E. Dynamic Process Limit Algorithm

On the basis of the discussion presently in the previous sections it is clear that a static BP limit cannot take care of all problems faced in the point of view of application loading and memory utilization. Different users might get different hit rates based on user access patterns that vary in the frequency and duration of application use.

```
lru_hit_rate ← calculate the latest hit rate from the number of hit in n slot
reclaimable_free ← free pages + remained page caches
if lru_hit_rate < TR_HLOW and pr_limit < MAX_PRLIMIT and
   reclaimable_free > TR_MEM
   then process_limit++;
if lru_hit_rate > TR_HHIGH and pr_limit > MIN_PRLIMIT
   then process_limit--;
```

Fig. 1. Background process limit based on hit rate [4]

1) When the hit rate decreases to TR_HLOW the algorithm increases the BA cache size by increasing the BP limit. The limit is not increased if the reclaimable free pages of the system are less than a threshold TR_MEM to maintain a limited amount of page cache.
2) When the hit rate goes beyond a threshold TR_HHIGH, the algorithm decreases the BA cache size by decreasing the BP limit. The free pages can then be used by the system to store pages of cached applications at the cost of LRU applications in the application cache.

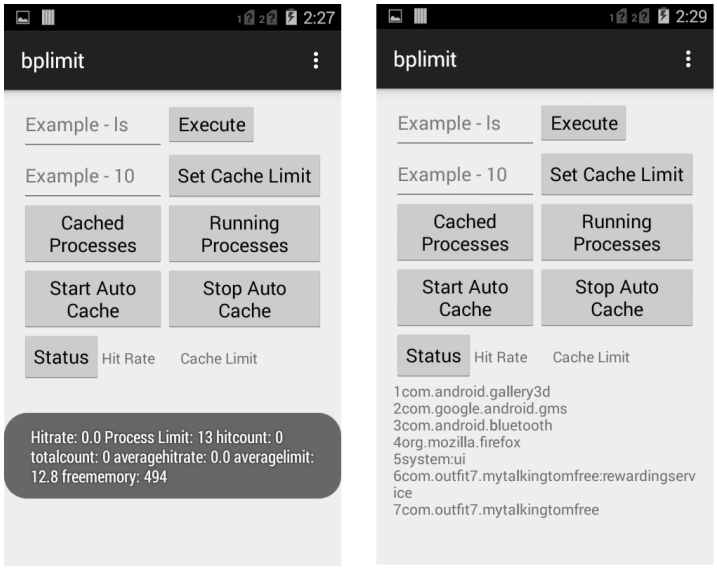

Fig. 2. Android Application

F. Machine Learning to set the heuristic parameters

To determine the threshold limits we need to know the long run usage pattern of the user, say 24 hours application launches-size and duration of applications used- in order to determine the threshold limits of background cached processes. Clustering can be used to form 2 clusters and the maximum threshold can be set to the size of the cluster that represents high usage applications. Minimum can be set based on the number of applications densely clustered. To set the threshold of free memory, average free memory can be determined in the 24 hour period. If this value is too big, the threshold can be set close to the oom threshold.

G. Aggregation of features

When the background cached process limit is decreased, then apps have to be deleted from the memory and when its increased new applications have to be loaded in memory. Android does this by keeping a LRU list of processes that have to be killed. Features like time spent and memory required by applications in foreground, background and cached state and recency can be gathered by the analysis of memory logs of past three hours and temporal usage probability of an app can be gathered by 2-4 days analysis of memory dumps. These can be used to decide the importance level of an application.

H. Score Calculation for apps of user interest

Usage based importance score can be calculated on the basis of the extracted features by fitting an exponential function or linear function to assign weights to the different features. It is important to note equal importance of each feature thereby suggesting the use of a linear decay function of weights for calculation of score. The weight of foreground

load = size*time usage of application and recency are most important features followed by background and cached loads of the application. Temporal probability calculated over a 4 day period is given the least importance because app usage patterns change on a daily basis.

score = recency* w1+ foreground load * w2+ background load * w3+cached load * w4+ temporal prob * w5

The weight w1 to w5 follow linear decay y=a *x + b with w1 being the largest and w5 being the smallest.

I. *Final Model*

The overall model that is implemented and described in section 5.3 to 5.9 can be summarized in the form of architecture diagram in figure 3. This integrates the work done in base paper with the improvement in the form of features and prediction of score of killing the cached application. The implementation is done as a sticky service on Android OS that polls for hit rate and aggregated features at intervals of 3 seconds and 1 minute respectively.

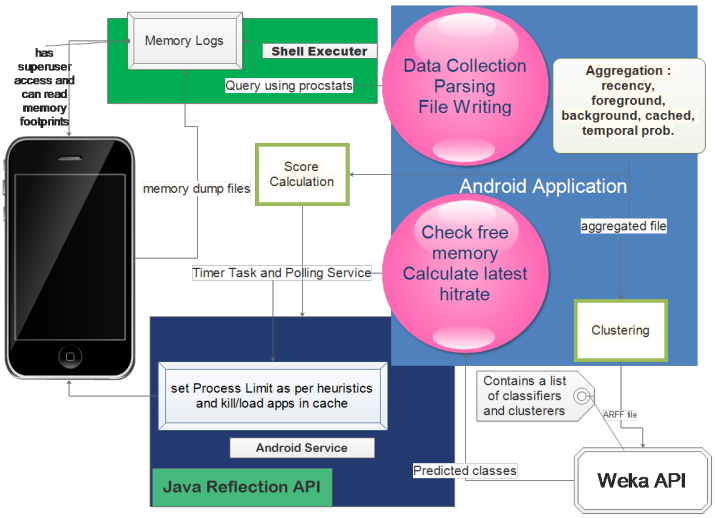

Fig. 3. System Architecture Diagram

IV. RESULTS

The efficiency of the features chosen and how effective the proposed methodology is when launching a set of applications can be benchmarked against the default scheme used by Android in terms of analysis of hit rate, average response time, number of kills of applications and average free memory maintained by the scheme.

A. *Experimental Setup*

The experiment of the algorithm is carried on the following configuration on a physical smartphone. The application is deployed as a system application with permission to read memory dumps, set background limit and kill processes in a superuser environment to eliminate the problem of signing an application with manufacturer's key.

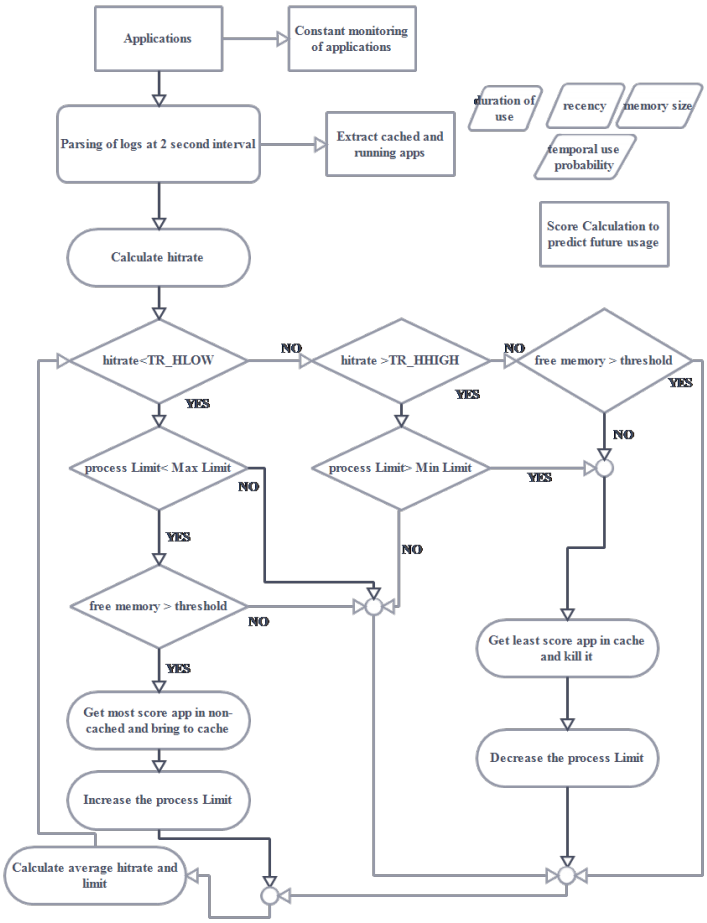

Fig. 4. Flowchart of proposed methodology

Smartphone configuration

OS: Android 4.4.4 (cyanogen mod)
RAM: 1 GB
Smartphone: Samsung Galaxy Grand I9082
Tool: monkey

The Monkey is a program that runs on emulator or device and generates pseudo-random streams of user events such as clicks, touches, or gestures, as well as a number of system-level events. You can use the Monkey to stress-test applications that you are developing, in a random yet repeatable manner. Delay after each event, number of events and packages to launch and interact with can be set using a command. 5 Runs lasting 20 minutes each are carried on to analyze the response time, number of kills and average memory on both the original scheme and our proposed model. To get the same number of events on both the approaches the same seed value can be passed on multiple runs. The numbers of application accesses may be different during different runs because monkey randomly performs events like clicks, typing, loading a file, switching an app and we cannot have control over it although percentages can be specified for access patterns. But, that can be biased run and hence random runs are made.

B. Hit rate and Process Limit

The above implementation on the Android platform tries to keep the hit rate in a dynamic zone of values improving the amount of page cache memory by restricting the number of background applications. Clustering of user activities in 24 hour duration helps to get the number of frequently accessed applications by the user. This helps to set the thresholds of the algorithm. The hit rate for most users is above 30% in all runs. When a few applications are required again and again by the user, the background process limit falls down to 5 processes whereas in diverse case it provides 33% efficiency improvement over the high threshold case (set to 10). Thus for all cases of application launching patterns there is an overall reduction in the number of background applications maintained by the operating system leading to more free memory and amount of memory available for page cache. The algorithm thus allows the system memory to be used effectively, without sacrificing the hit rates of the BA cache.

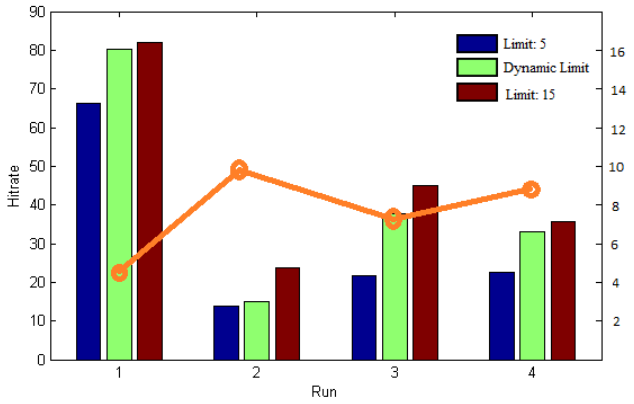

Fig. 5. Hit rate for 4 runs of the algorithm

C. Average Response Time

TABLE I. RESPONSE TIMES

	Normal		Modified	
Runs	**Total**	**Average**	**Total**	**Average**
Run 1	232.9	2.74	151.3	1.78
Run 2	176.3	2.05	134.16	1.56
Run 3	193.6	2.2	145.2	1.65
Run 4	187.92	2.16	140.07	1.61
Run 5	151.7	1.85	117.26	1.43

Response time is defined as the time taken from touching to complete launching of app with UI components. The cumulative average response time after round 3, 4 and 5 are 2.33, 2.2875, and 2.2 for the normal case and 1.66, 1.65, and 1.60 for the modified case respectively. So there is 28.27% reduction in average response time of launching the apps.

D. Number of Kills

The number of kills has drastically reduced from approx. 36% to 12% in the modified algorithm showing the caching behaviour of heavy usage applications by the user in the main memory.

TABLE II. NUMBER OF KILLS

	Run 1		Run 2		Run 3		Run 4		Run 5	
Algorithm	**A**	**K**	**A**	**K**	**A**	**K**	**A**	**K**	**A**	**K**
Normal	85	30	86	33	88	38	87	36	82	31
Modified	85	10	86	12	88	11	87	16	82	10

Legends: A: Accesses, K: Kills

E. Average Free Memory

The recorded average free memory in the runs is: 310, 352, 410, 374 and 328 in the modified case and 420, 436, 445, 429 and 421 in the original case. Clearly less free memory is available when the modified algorithm during the runs but that helps in caching more applications in the background leading to less response times.

V. CONCLUSION

With proper setting of the background process limit and retaining applications of user interest most of the time helps in bringing down effective application loading time of opening a limited number of applications and this scheme is very beneficial to access patterns that do not change rapidly with time which is the real world case. This is due to the fact that the number of kills for frequently used applications is decreased. Average free memory is reduced on the device which is a very direct result of extensive caching. If the same cluster of applications will be reused again and again then definitely the above research will help in better loading times of applications hence improving the overall experience of a smartphone user.

REFERENCES

[1] Android Blog, XDIN. [Online] Available at: $http://www.androidenea.com/2009/06/android-boot-process-from-power-on.html$ [Accessed: 7 October 2014]

[2] RECEIVE BOOT COMPLETED, StackOverflow. [Online] Available at $http://stackoverflow.com/questions/5051687/broadcastreceiver-not-receiving-boot-completed$ [Accessed: 7 October 2014]

[3] Cheng-Zen Yang; Bo-Shiung Chi, "Design of an Intelligent Memory Reclamation Service on Android," Technologies and Applications of Artificial Intelligence (TAAI), 2013 Conference on , vol., no., pp.97,102, 6-8 Dec. 2013

[4] Baik, K.; Jaehyuk Huh, "Balanced memory management for smartphones based on adaptive background app management," Consumer Electronics (ISCE 2014), The 18th IEEE International Symposium on , vol., no., pp.1,2, 22-25 June 2014

[5] H. Falaki, R. Mahajan, S. Kandula, D. Lymberopoulos, R. Govindan, and D. Estrin, Diversity in Smartphone Usage, in Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services (MobiSys 10), 2010, pp. 179194.

[6] Jain, R.; Bose, J.; Arif, T., "Contextual adaptive user interface for Android devices," India Conference (INDICON), 2013 Annual IEEE , vol., no., pp.1,5, 13-15 Dec. 2013

[7] Android application (performance and more) analysis tools[Online] Available at: $http://www.vogella.com/tutorials/AndroidTools/article.html$ [Accessed: 7 March 2015]

[8] Managing Your App's Memory, Android Developers.[Online] Available at: $http://developer.android.com/training/articles/memory.html$ [Accessed: 7 March 2015]

[9] "Tutorial For Android." : Access Internal Classes in Android. [Online] Available at: $http://www.tutorialforandroid.com/2010/07/access-internal-classes-in-android.html$ [Accessed: 7 March 2015]