International Journal of Advanced Information Science and Technology (IJAIST) ISSN: 2319:2682
Vol.3, No.12, December 2014 DOI:10.15693/ijaist/2014.v3i12.80-87

A New Approach to Optimal Memory Partitioning Technique in Android Mobile Platform

Jadhav Manoj
SCSE, VIT University
Vellore(India)

Bajaj Sarveshwar
SCSE, VIT University,
Vellore(India)

Pratik Patrikar
SCSE, VIT University
Vellore(India)

Abstract - App-market is a type of digital distribution platform to launch newly developed application software, which is delivered as a component of an operating system on a tablet or smartphone. Application store contributes to run numerous applications in mobile platform. Internal memory is a very limited resource particularly in embedded systems like tablets and mobiles. Out of memory killer (OOMK), Activity service manager (ASM) and Low memory killer (LMK) is widely memory management techniques used in mobile platform. They forcibly terminate the application if physical memory becomes insufficient. Memory shortage incurs thrashing and fragmentation, thus slowing down the application performance. In this paper, we have proposed well-organized memory partitioning technique that resolves degradation of existing application life cycle induced by LMK, ASM and OOMK. This paper proposed the complete conception of virtual memory nodes in operating systems of Android devices.

Keywords: Low memory killer, Activity Service Manager, Out of memory killer.

1. INTRODUCTION

Now days, many mobile phone user's uses built-in software or applications that the manufacturer includes into the mobile phone as well as the third-party applications downloaded from various app-markets. The App-market or app-store is a type of digital distribution platform designed to release application software. In those types of systems, the memory consumption of the third-party applications leads to insufficient memory space to run those applications resourcefully. As we know, low–end devices don't have sufficient memory so memory shortage may occur repeatedly. The term Memory management in mobile devices is very essential because the devices have relatively small memory capacity with no ad-hoc expansion, and the memory management of downloaded applications cannot be controlled or tested at the time of manufacturing. To cope up with the memory shortage, low memory killer (LMK) [1], [2]-[3] is the most widely adopted solution. In case of memory shortage, it forcibly terminates less important applications until the operating system (OS) secures enough free memory space to run a new application. The list of the order of application importance is managed by user-space daemons, like thread manager and an activity manager. The activity manager acts as a traffic controller for the overall activities (e.g. foreground and background processes and system resources) running on the mobile device. The functionality of the activity service manager is to receive the request from user and handle them. The frequent operations of LMK and out-of memory killer (OOMK) could seriously degenerate user perceived performance in two ways. First, because the memory space of a victim application [4] is unloaded, the unloaded memory should be reloaded at the next launching of the victim application and it could slow down the application performance. To select a victim application, OS considers the following criteria: the number of threads, the CPU running time, the scheduling priority of victim, and whether it directly accesses the hardware or not. Second, the built-in applications, such as Phone, short message service (SMS), and Contacts, can be forcibly terminated.

When memory shortage occurs frequently the page fault induced and it leads to increase in the cost of page replacement. It makes application more prone to miss to require deadline [5] [6] and as a result encounters thrashing [7]. Consequently, a user experiences slow down performance even for built in application.

In this paper, we have proposed new memory partitioning technique to improve application performance which saves the efforts of Out-of-Memory Killer (OOMK) and Low Memory Killer (LMK). We explained new memory partitioning at the OS level, which limits the page reclamation within the partitioned memory range based on the well-defined hierarchy importance of applications. The application hierarchy is

International Journal of Advanced Information Science and Technology (IJAIST) ISSN: 2319:2682
Vol.3, No.12, December 2014 DOI:10.15693/ijaist/2014.v3i12.80-87

classified into built-in applications, applications from trusted sources, and unknown applications from untrusted sources.

2. LITERATURE SURVEY

The operating system generally supports the feature likes page reclamation [8], swap in/out [2], OOMK [3], Activity Manager Service (ASM) and Low Memory Killer (LMK) to secure free memory space under insufficient memory. There are some issues involved in implementation of those features:

2.1 Page Reclamation and swap in/out mechanism:

This mechanism uses heuristic methods to find victim process among the available and helps to obtain available memory in the system. However, it finds target pages based on least recently used page (LRU) replacement algorithm. It blindly handles all the processes without the platform level semantics, which are important systems application in mobile platform.

The swap in/out mechanism [9] is widely used to run applications that require larger memory than the physical memory capacity. Because swapping operations work with a slow storage device with limited durability, they fail to provide reasonably predictable performance [10][11].So, swap in/out mechanism is not used in most of the mobile device manufacturers [12][13].

2.2 Memory management of OOMK

OOMK endeavours to overcome the memory shortage from the out-of-memory status by terminating a lower priority process. The original role of OOMK is to kill unimportant processes based on the memory score of processes heuristically when the memory capacity is deficient. OOMK calculates the score using following heuristics:

1) High score is assigned to process who occupied large memory space.
2) Low score is assigned to the process that runs for longer time.
3) Killed the process which have large no of child process.
4) Do not kill super user's process.

In case of embedded operating systems like android, we need to assign priority according to their repeatedly accessed applications. As OOM killer do not give any priority to frequently accessed applications, it simply kills the processes even though enough memory space available in the system. It proves that killing the processes based upon some heuristics leads to unfairness in order of killing.

However, the operation of OOMK seriously degrades the execution speed of new applications due to the *thrashing* [9]. When a new application is launched under a high memory pressure, OOMK forcibly terminates a process based on the relative severity in order to retrieve additional memory space. OOMK attempts to retrieve the available memory by killing the processes of the lower memory score as a victim process to avoid an out-of-memory situation. It heuristically determines the victim processes according to the number of execution frequencies of the application, the execution time of the application, the scheduling priority of the process, the application to access devices, and the application authorized by the root user.

2.3 Low Memory Killer Management:

OOM killer does not give any priority to frequently accessed applications and during the lifetime of process priority value is set as static. Because of limitations of flash memory, Android doesn't have swap space. Android main memory contains so many empty processes, so order of killing should be different unlike OOM killer. So, Low Memory Killer is another approach used in Android to avoid some of the problems in OOM killer. The existing mobile platforms manage the memory Management of the applications in a single memory space. These applications mainly consist of built-in applications by the manufacturer and external applications downloaded from the application store by the user.

The original role of LMK is to automatically terminate the applications in an LRU list [1]-[3] when the available memory reaches a specified threshold of the system. The operating system starts to kill the oldest unneeded processes in the LRU list to retrieve the free memory space for the execution of new applications. If the system reaches the threshold of free physical memory, LMK terminates the applications that are relatively less important among the running applications.

However, the memory fragmentation gradually increases because the operating system reclaims the memory blocks of unimportant processes with the unit of page from a physical memory. As the more memory fragmentation occurs, the small size memory blocks gradually increase further, which leads to an additional memory management costs such as the merging

of small blocks by a memory allocator, the time required to read all the non-adjacent memory blocks at once, and the scheduling cost between the memory blocks because of the formation of too many small blocks. These too many small blocks increase the memory scheduling cost to determine whether the higher priority processes are waiting or running during the allocation and the release of the small blocks in the pre-emptive operating system.

2.4 Causes Of Low Memory Circumstances:

There are many reasons; those leads to low memory situations, some of the reasons are availability of empty application in memory, infrequently accessing applications in memory and unwanted system applications into memory. This section focuses on different types of reasons that cause to low memory scenarios

A. Empty Process Management:

In Linux, whenever the application is terminated by user all associated information related to that application is removed from main memory. But in android the associated information is not removed from main memory, because if user accesses same application in future we do not need to load it again from secondary memory

The figure represents the list of empty applications along with their states.
The advantages and disadvantages of empty applications are as follows.

Advantages of Empty Applications:

1. Reduces response time due to decreasing the loading time of applications available in the main memory.
2. We can save the power consumption, if we avoid too many loads of application from secondary memory.

Disadvantages of Empty Applications:

1. It increases the response time of applications which are not there in main memory.
2. Too much of empty applications lead to repeated low memory scenarios.

B. Loading System Apps:

Generally, there are two types of applications are available in Android. They are system applications (installed in/system/app folder) and third party applications (installed in/data/app folder). To access internal resources, System application has more privileges than third party application. During the boot time ASM loads some of system applications like phone, contacts, email, and messages etc. in to the main memory irrespective of their usage.

Problems with ASM:

i. ASM does not consider user infrequently accessing applications for killing and it does not consider free memory size while killing.
ii. Irrespective of free memory available ASM kills applications based on LRU list. From ASM kills applications before low memory scenarios will occur, because it does not allow number of empty applications more than some threshold. So we need consider free memory available in main memory and to do not kill applications aggressively. Finally we can reduce number of applications gets killed in a period of time.
iii. ASM does not consider an application is interested to user or not before killing that Application. Because it simply kills application based on LRU and it does not consider the frequency of that application. If ASM kills user interested applications then we need to reload such application. So if we can predict the user future accessing applications based on frequency. Finally, we can protect such kinds of applications get killed in low memory scenarios, and then we can reduce number of applications get killed and number of load operation.

3. SYSTEM DESIGN:

This section describes the design and implementation details of the proposed system. The implementation of optimal memory partitioning will be done at OS level to solve the problem of insufficient physical memory for the applications

International Journal of Advanced Information Science and Technology (IJAIST) ISSN: 2319:2682
Vol.3, No.12, December 2014 DOI:10.15693/ijaist/2014.v3i12.80-87

which results in poor user responsiveness of inbuilt applications in mobile phones. This technique dynamically sets the memory layout at boot time based on permission privilege given to the root user, which supports various mobile devices range in between from low-end and high-end. The technique named as a Virtual Markov memory Node (VMNODE) which is enhancement of Virtual memory Node followed by Markov Algorithm applied on it.

3.1 Design of VMNODE:

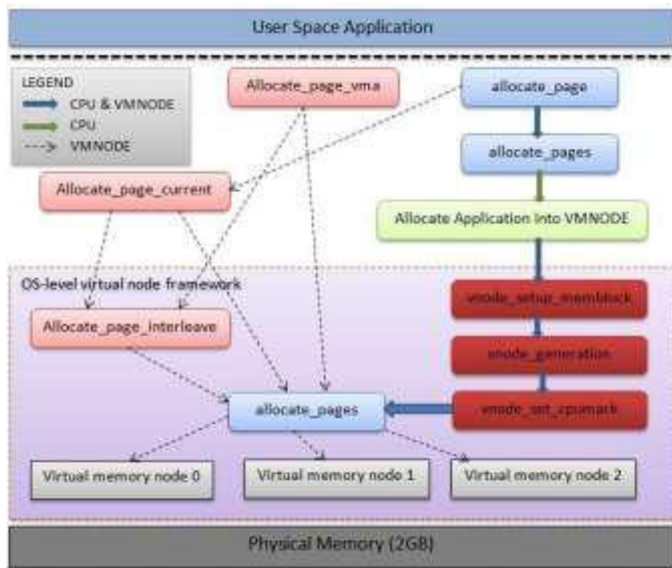

Figure 1.Architecture of optimal memory partitioning

Figure 1. shows the overall architectural design of proposed optimal memory partitioning technique for the mobile applications that have limited memory space. The new optimal memory partitioning technique consists of following three components:

1. *vnode_setup_memblock* it manages the mapping between the physical memory address and a virtual node to separate the physical memory
2. *vnode_generation* it generates the specified no. of virtual memory nodes from physical memory node and also determines the size of table to hold the address range of physical memory
3. *vnode_set_cpumask*It allocates the CPU masks for mapping between a virtual memory node and a specified CPU.

VMNODE has following two main advantages for mobile phones with limited memory capacity:

1. Memory Isolation: as it splits physical memory, VMNODE controls unnecessary consumption of memory by untrusted applications. e.g. VMNODE0 for trusted applications i.e. built in apps and VMNODE1 for untrusted applications like malware software and memory hog software, and
2. Reduction in the no. of LMK/OOMK operations: VMNODE minimizes the frequency of LMK/OOMK operations whenever memory shortage occurs.

The idea is to allocate and release the memory area of the applications in physical memory area to avoid unknown applications from untrusted resources. Memory techniques like on demand paging, page reclamation, and page defragmentation does the task of memory allocation and de-allocation and release of the applications. This means that the core built in applications encounters the performance degradation more often because of unknown applications from the untrusted sources, given below:

1. Thrashing: it affects the execution time of the applications due to page fault and page replacements.
2. Memory Fragmentation: it increases the cost of maintaining too small memory fragments which in turn increases the scheduling cost while allocating or releasing the scattered small memory fragments.

With this proposed idea, the operating system can reduce the cost of thrashing and fragmentation of physical memory. The virtual memory partitioning technique protects untrusted applications from harming the execution time of core built in applications.

The existing system is not able to determine a page boundary region to reclaim pages because the existing approach maintains the memory's usage based on the amount of memory of the processes without the virtual memory access area. Therefore, the operating system can be equipped ad implemented with a mechanism to allocate the memory pages of the processes such as the virtual memory nodes that appear to be a physical memory.

- ✓ The *allocate_page_vma* function shown in Figure 1 manages the pages of the applications in the virtual memory space. It connects the memory pages of the process to the *allocate_page_interleave* function.
- ✓ The *allocate_page_interleave* function executes the low-level operation to interconnect an application and a memory area. If the operating system needs to find the allocated memory address currently according to the process request
- ✓ The*allocate_page*functioncallsthe *allocate_page_inteleave* function via the *allocate_page_current* function.
- ✓ At the end *_allocate_pages* function allocates/releases the memory area of the process using the processing result of VNODE's three components:

1) *vnode_setup_memblock*,
2) *vnode_generation*, and
3) *vnode_set_cpumask*.

3.2 INTELLIGENT MEMORY MANAGEMENT SERVICE

90% of the smartphone users have stationary usage patterns for session lengths. Some empirical studies have shown that many people have their regular user behavior patterns of using smartphones and mobile phones. From this aspect, more accurate memory reclamation can be performed on the basis of these regular temporal patterns. In this paper, we are presenting the design of an intelligent memory reclamation service to tackle this memory management issue using the Markov Decision Process (MDP) model [5] to decide the reclamation priorities of activities based on the user behavior patterns. The MDP model has shown its prominence in dealing with many dynamic decision-making problems, such as power saving [6] and intrusion detection [7], by recognizing system behavior patterns at run time. In the proposed system service, MDP predicts the survival probabilities of the running activities which reside in the system *stop queue*. The reclamation priorities of running activities are then decided according to their reclamation rewards, which are the products of their survival probabilities and their allocated memory space. When the volume of the free memory space is lower than the user defined threshold, activities with high priorities will be killed to release their memory space. Therefore, the proposed memory reclamation service can reduce the probability of killing an activity that will be re-invoked in a very short time.

The proposed intelligent memory reclamation service employs an MDP model to learn the stationary user behavior patterns and automatically kill activities with high reclamation rewards. Since the user behavior patterns of user are learned, the MDP-based service can reduce the possibility of erroneously killing applications that should actually be executed in memory. This section will first introduce the system operation model of the MDP-based service. Then, the details of the MDP-based prediction scheme are described.

A. System Model

Figure 2 shows the system operation model for the proposed MDP-based memory reclamation service. When an activity executes its onStop() callback method, it will enter the *stop queue* in Android. For deciding which *Stopped* activity can be killed to return memory space back to the system, the MDPbased service periodically inspects the execution status of each in-memory activity and calculates its reclamation reward *Rr* for the next inspection.

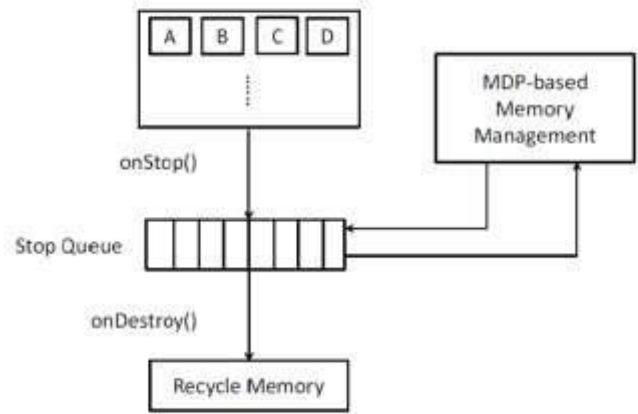

Figure. 2. System model in the intelligent memory management service.

The reclamation reward *Rri* of an background running activity *acti* is decided according to two factors: its survival probability at the next inspection (*Psi*), and its allocated memory size (*mi*). Since Android maintains the stop queue to keep track of activities having entered the *Stopped* state, the MDP-based service calculates the survival probability of each activity according to the historical queuing time in the stop queue. With the MDP-based service, Android now has two different situations to reclaim memory from the activities in the system. For the first case, the MDP-based service proactively decides whether it needs to trigger the reclamation process during the inspection according to the volume of free memory space (*Mf*) and a user-defined threshold (*Mu*). If the MDP based service finds *Mf* ≤ *Mu*, it will start the MDP-based reclamation process and continues to kill the applications with high reclamation rewards until *Mf* > *Mu*. However, between two inspections the MDP-based service cannot make the reclamation decision and the user may suddenly launch applications to consume much free memory space. In this case, the build-in LRU-based reclamation scheme will be invoked when *Mf* is lower than the system-defined threshold *Ms*. Therefore, two thresholds *Mu* and *Ms* control the initiation timings of the MDP-based reclamation and the LRU-based reclamation.

B. MDP-based Prediction

To learn the user behavior patterns, the kernel of the proposed intelligent memory reclamation service is constructed using the Markov Decision Process (MDP) model [5]. For a given finite discrete-time fully apparent MDP model, characteristics of the system are observed and recorded as the element states at each discrete time. The MDP is thus a tuple _*S, A,D, T,R*_, where *S* is the state space, *A* is the action set, *D* denotes the set of the decision points at time 99 steps, *T* (*si, aj, si*+1, *tk*) is a transition function specifying the probability of going to state *si*+1 from *si* if action *aj* is executed at time *tk*, and *R*(*si, aj, si*+1, *tk*) is the reward function to obtain the reward as the action *aj* is executed at *tk*

when the system state goes from si to $si+1$. The memory reclamation problem in this work is modeled as a three-state MDP problem: $s0$ in which Android has enough free memory space ($Mf > Mu$), $s1$ in which the free memory space size is lower than a user-defined threshold Mu but larger than the system-defined threshold Ms ($Ms < Mf \leq Mu$), and $s2$ in which the free memory space size is lower than the system-defined threshold Ms ($Mf \leq Ms$). If Android is in state $s1$, the MDP-based service will start the MDP-based reclamation process in its periodic inspection such that the system can go back to state $s0$. If Android is in state $s2$, the build-in LRU-based reclamation scheme will be invoked to keep the system safe such that the system can go back to state $s1$.

3.3 Slab Algorithm

For handling low memory (bytes) requests, we need an efficient algorithm which supports less fragmentation and less time while initializing an object. Slab algorithm reduces internal fragmentation, inside pages and within frames. It works on the concepts of Buddy algorithm. Also it is used for handling memory requests for small bytes [3]. Slab allocator in Linux creates a cache for each object of distinct sizes. It maintains a list of caches for frequently accessed and used kernel data structures like inode, task struct, mm struct etc. Cache is a collection of slabs. Slab contains the pages whose size may be one or two pages. Slab contains a 19 group of similar type of objects. Fig 3 depicts the clear idea of slab allocator. It shows the relation among caches, slabs and objects.

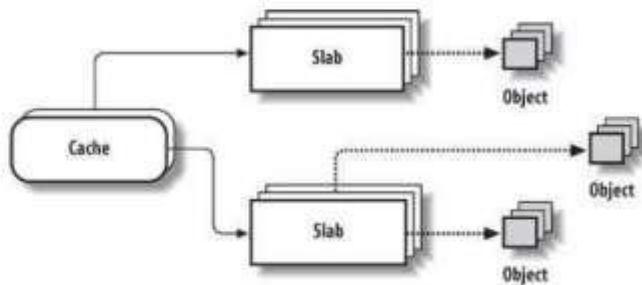

Figure 3: Slab allocation in Linux [5]

1. Slab Data Structures: Cache descriptor depicts the type of object that could be cached. Slab keeps a list for all the caches. Slab descriptor holds the pointers to actual memory address of the object. Every slab has some state like full, partial, and empty. Object descriptor holds two things; first one object is free and holds a pointer to next free object and second one holds contents of object.

2. Methods for allocating and de-allocating objects: Objects are allocated using kmem cache alloc(cacheptr), where cacheptr points to the cache from which the object must be obtained. Objects are released using kmem cache free(cacheptr,objjp).

4. PROPOSED WORK

The arrows in Figure 4 show the operating structure between CPU and memory. The root user can adjust the generation procedure of the virtual memory nodes at boot time. For example, it will be assumed that for the memory layout, the trusted applications can be made to run in VMNODE0 and the untrusted applications can run in VMNODE1.

In the mobile devices, the definition of the typical two types of software is as follows:

1) Trusted applications: These are the built-in applications and downloaded applications from trusted sources.
2) Untrusted applications: These are downloaded applications from untrusted sources. An untrusted application potentially encompasses malicious code, memory hog, high power consumption, and unnecessary CPU usage. Abnormal system behavior and system reboot mostly results from these applications.

By adopting the proposed approach, the operating system can controls the applications to avoid reaching memory shortage while running the applications. The proposed memory partitioning technique settles the problem of the single memory space by running the trusted applications within VMNODE0 only. That is, the built-in applications from the trusted sources stay in the memory until users directly exit their applications, as shown in Figure 4.

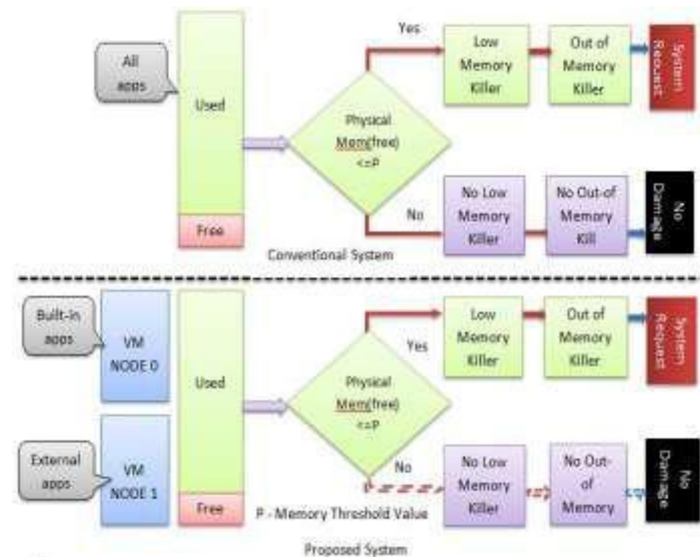

Figure. 4. This flow diagram describes the operation of LMK/OOMK for the trusted applications and the untrusted

applications. The external applications are the untrusted applications that run in VMNODE1. 'T' denotes the trusted applications of the official application store, and 'U' denotes the the untrusted applications of the unofficial application store. The threshold of free physical memory is 76 MB.

As mentioned in above section, the Markov Decision Process (MDP) model [5] decides the reclamation priorities of activities based on the user behavior patterns. After applying the given scenario again if the system is facing the same memory shortage problem and the OS decides to kill trusted applications. To overcome this problem, for trusted applications assigned to VMNODE0 we can apply the Markov Decision Algorithm for calculating the least recently used applications from the users observed behavior patterns. The procedure how Markov Decision algorithm works is described in Section 3.2 A. Along with it to avoid less fragmentation we can apply Slab algorithm so that the problem of too many small memory fragments can be resolved and we can have the big enough room to accommodate the newly loading applications. Section 3.3 describes the details of Slab Algorithm.

5. PROPOSED ALGORITHM

1. **Design of VNODE**
   1.1 vnode_setup_memblock
   mapping of physical memory with virtual
   node 1.2 vnode_generation
   generation of specified no. of virtual memory nodes from physical memory node and to determine the size of the table for holding the address range of the physical memory.
   1.3 vnode_set_cpumask
   allocating the CPU masks to support mapping between a virtual memory node(s) and specified CPU(s) to recognize CPU-Hotplug and CPU-DVFS enabled multicore environments

2. **Assignment of VNODE** VMNODE0
   for built-in applications. VMNODE1
   for trusted applications
   VMNODE2 for untrusted applications such as Malware software, memory hog.

3. **Management of the pages of the applications in the virtual memory space**

4. **Finding the allocated memory address of untrusted applications**

5. **Allocation and release of memory area by killing the untrusted applications**

6. **Implementation of Markov Decision algorithm for killing the applications**

7. **Implementation of Slab Algorithm**

6. CONCLUSION

The conventional memory management features frequently induce thrashing, page fault, and page replacement to secure free memory. The proposed optimal memory partitioning technique inhibits the performance degradation of applications caused by thrashing, frequent page faults, and page replacements. In addition, the proposed approach supports complete virtual memory isolation based on a dis-contiguous memory access model to separately run applications from the trusted sources and the untrusted sources. Markov Decision Process model helps to decide reclamation priorities of activities based on user behaviour pattern. It drastically reduces the number of LMK/OOMK operations by reducing the number of page faults and page replacements and also the recently used applications are not killed. Slab algorithm also helps to improve the problem of too many small fragments which reduces the cost of merging of memory holes as it works on concept of Buddy algorithm. Consequently, the proposed approach overcomes the low performance of the trusted applications induced by LMK/OOMK operations trusted applications induced by LMK/OOMK operations during memory shortage.

7. REFERENCES

[1] G. Lim, C. Min, and Y. I. Eom, "Enhancing application performance by memory partitioning in Android platforms," *in Proc. IEEE International Conference on Consumer Electronics*, Jan. 2013.

[2] S. Nomura, Y. Nakamura, T. Hattori, K. Nagata, and S. Yamaguchi, "Managing process memory size in smartphone," *in Proc. ComputerSystem Symposium*, Dec. 2012.

[3] J. Kook, S. Hong, W. Lee, E. Jae, and J. Kim, "Optimization of out of memory killer for embedded Linux environments," *in Proc. ACM Symposium on Applied Computing*, pp. 633-634, 2011.

[4] S. Jiang and X. Zhang "Adaptive page replacement to protect thrashing in Linux," *in Proc. Annual Linux Showcase & Conference*, Nov. 2001.

[5] E. Lee, K. Koh, and H. Bahn, "Dynamic memory allocation for real-time and interactive jobs in mobile devices," *IET Electronics Letters*, vol. 46, no. 6, pp. 401, Mar. 2010.

[6] S. Jung, Y. Lee, and Y. H. Song, "A process-aware hot/cold identification scheme for flash memory storage systems," *IEEETransactions on Consumer Electronics*, vol. 56, pp. 339-347, May 2010.

[7] M. Lin, S. Chen, G. Lv, and Z. Zhou, "Optimised Linux swap system for flash memory," *IET Electronics Letters*, vol. 47, pp. 641-642, May 2011.

[8] R. Prodduturi, "Effective handling of low memory scenarios in Android," Indian Institute of Technology, Mar. 2013.

[9] O. Kwon and K. Koh, "Swap space management technique for portable consumer electronics with NAND flash memory," *IEEE Transactions on Consumer Electronics*, vol. 56, pp. 1524-1531, Aug. 2010.

[10] K. S. Yim, H. Bahn, and K. Koh, "A flash compression layer for SmartMedia card systems," *IEEE Transactions on Consumer Electronics*, vol. 50, pp. 192-197, Feb. 2004.

[11] Y. Feng and E. D. Berger, "A locality-improving dynamic memory allocator," *in Proc. Workshop on Memory System Performance*, pp. 68-77, 2005

[12] S. Park, H. Lim, H. Chang, and W. Sung, "Compressed swapping for NAND flash memory based embedded systems," *in Proc. International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation*, pp. 314-323, Jul. 2005.

[13] O. Kwon, Y. Yoo, K. Koh, and H. Bahn, "Replacement and swapping strategy to improve read performance of portable consumer devices using compressed file systems," *IEEE Transactions on Consumer Electronics*, vol. 54, no. 2, pp. 551-559, May 2008.

AUTHORS BIOGRAPHIES

Jadhav Manoj received the B.E. degree in Computer Science and Engineering from Walchand Institute of Technology, Solapur, Solapur University, Solapur India in 2012. He is pursuing the M.Tech. in Computer Science and Engineering from VIT University, Vellore, India. His research interest includes Operating Systems and design, Big Data Analysis and cloud Computing.

Pratik S. Patrikar received the B.E. degree in Computer Science and Engineering from Sant Gadge Baba Amravati University, Amravati India in 2014. He is pursuing the M.Tech. in Computer Science and Engineering from VIT University, Vellore, India.

Bajaj Sarveshwar received the B.E. degree in Computer Science and Engineering from MMCOE, Pune, Pune University, Pune, India in 2013. He is pursuing the M.Tech. in Computer Science and Engineering from VIT University, Vellore, India.