Policy-extendable LMK filter framework for embedded system

LMK: the state of the art and its enhancement

Kunhoon Baik
Samsung Electronics Co., Ltd.
Korea Advanced Institute of Science and Technology
knhoon.baik@samsung.com, knhoon.baik@kaist.ac.kr

Jongseok Kim
Korea Advanced Institute of Science and Technology
paldad@kaist.ac.kr

Daeyoung Kim
Korea Advanced Institute of Science and Technology
kimd@kaist.ac.kr

Abstract

Background application management by low memory killer (LMK) is one of the outstanding features of Linux-based platforms such as Android or Tizen. However, LMK has been debated in the Linux community because victim selection mechanism with a specific policy is not suitable for the Linux kernel and a flexible way to apply new policies has been required. Thus, several developers have tried implementing a userspace LMK like the ulmkd (userspace low memory killer daemon). However, not much work has been done regarding applying new polices.

In this paper, we present a policy-extendable LMK filter framework similar to the out-of-memory killer filter discussed at the 2013 LSF/MM Summit. The framework is integrated into the native LMK. When the LMK is triggered, each LMK filter module manages processes in the background like packet filters in the network stack. While the native LMK keeps background applications based on a specific policy, the framework can enhance background application management policy. We describe several benefits of the enhanced policies, including managing undesirable power-consuming background applications and memory-leaking background applications. We also present a new LMK filter module to improve the accuracy of victim selection. The module keeps the applications which could be used in the near future by predicting which applications are likely to be used next from the latest used application based on a Markov model.

We implemented the framework and the module on Galaxy S4 and Odroid-XU device using the Linux 3.4.5 kernel and acquired a preliminary result. The result shows that the number of application terminations was reduced by 14%. Although we implemented it in the kernel, it can be implemented as a userspace daemon by using ulmkd. We expect that the policy-extendable LMK filter framework and LMK filter will improve user experience.

1 Introduction

Modern S/W platforms for embedded devices support a background application management. The applications stacked in the background are alive until the operating system meets a specific condition such as memory pressure, if a user does not kill the applications intentionally. The background application management permits fast reactivation of the applications for later access [2], battery lifetime can be longer because energy consumed by applications re-loading can be reduced. [13]

However, applications cannot be stacked infinitely in the background because memory capacity is limited. Instead, the operating system needs to effectively manage the background applications in low memory situation. To handle such a situation, the operating system provides out-of-memory handler. Unfortunately, it causes significant performance degradation to user interactive

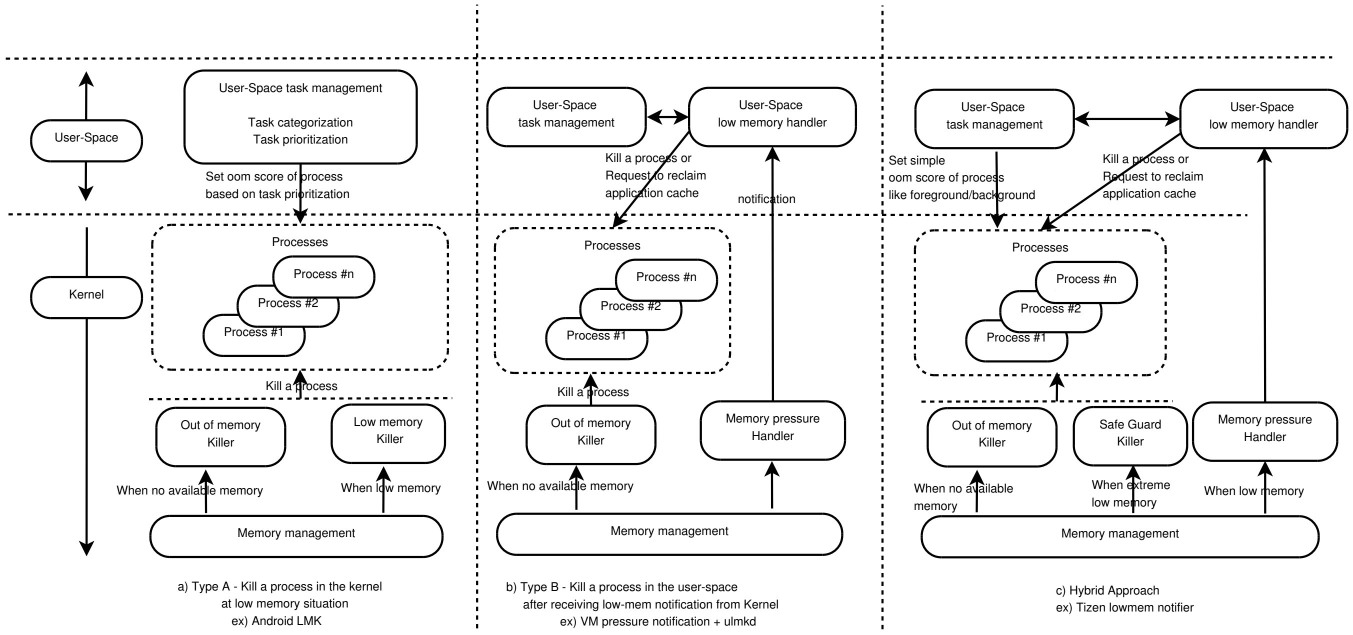

User-Space
Kernel
User-Space task management
Task categorization
Task prioritization
Set oom score of process
based on task prioritization
Processes
Process #n
Process #2
Process #1
Kill a process
Out of memory Killer
Low memory Killer
When no available memory
When low memory
Memory management
a) Type A - Kill a process in the kernel at low memory situation ex) Android LMK
User-Space task management
User-Space low memory handler
Kill a process or Request to reclaim application cache
notification
Memory pressure Handler
b) Type B - Kill a process in the user-space after receiving low-mem notification from Kernel ex) VM pressure notification + ulmkd
Set simple oom score of process like foreground/background
Safe Guard Killer
When extreme low memory
c) Hybrid Approach ex) Tizen lowmem notifier

Figure 1: Architecture according to types of low memory killer

applications by excessive demand paging because the operating system triggers the OOM killer under desperately low memory conditions.

To avoid excessive demand paging, it was necessary for the operating system to trigger a low memory handler before falling into desperately low memory conditions. Several approaches have been introduced [10, 12, 6, 15, 16, 11]. The approaches can be categorized into two types from a structural perspective. Figure 1 shows the operation flow according to the types of low memory handler. The type-A approach, which is shown in Figure 1-(a), is to kill a process with the lowest priority in the kernel, according to the process priority configured in the user-space when a low memory handler is triggered. The type-B approach, which is shown in Figure 1-(b), is to kill a process after prioritizing processes in the user-space when a low memory handler is triggered. The type-A approach is hard to change the priority of a process in a low memory situation, and the type-B approach suffers from issues like latency until a process is killed after a low memory handler is triggered.

The Android low memory killer (LMK) is one of the type-A approaches, and it has been used for a long time in several embedded devices based on the Android platform. However, to apply a new victim selection policy, the user-space background application management must be modified, and it is impossible to re-prioritize the priority of processes in a low memory situation. Therefore, type-B approaches like userland low-memory killer daemon (ulmkd), have received attention again because the kernel layer provides simple notification functionality, and the approach can give the user-space an opportunity to dynamically change a victim selection policy. However, the user-space LMK is unlikely to handle a low memory situation in a timely way for a case of exhausted memory usage. Although related developers have made several attempts, the user-space LMK still has unresolved issues. [11] Thus, it is still too early to use the user-space LMK to dynamically apply a new victim selection policy.

While the type-B approach has been actively discussed and developed, applying new victim selection polices has not progressed, even though an advanced LMK victim selection policy would improve user experience and system performance. In the case of smartphones, most S/W platforms adopt the least recently used (LRU) victim selection algorithm to select a victim application in a low memory situation. However, the LRU victim selection algorithm sometimes selects applications to be used in the near future because the applications is likely to depend on the last-used application, hour of day, and location. [9] Likewise, the LRU victim selection algorithm does not preferentially select undesirable applications, such as memory-leaking applications, in a low memory situation. [13] Thus, if a system provides a mechanism to easily apply a new victim selection policy, it would improve the user experience and system performance.

In this paper, we propose an LMK filter framework to provide such a policy extension. To do that, we suggest a new architecture, modifying the type-A approach, because the type-B approach is still difficult to apply in a commercial device. We re-factored Android LMK to solve the limitation of the type-A approach. Based on the re-factored Android LMK, we created an LMK filter framework to provide real-time policy extension. The LMK filter framework provides the interfaces for managing policy modules with a policy extension, and the engine for applying the new policy of the policy modules. In addition, we present a task-prediction filter module to improve the accuracy of victim selection. With the results provided, we expect that the LMK filter framework and the module will improve user's experience.

The rest of this paper is organized as follows. Section 2 briefly describes previous approaches to memory overload management in the Linux kernel. Section 3 describes previous studies for applying a new victim selection policy. Section 4 provides the LMK filter framework and detailed implementation. Section 5 provides the task-prediction filter module to enhance the accuracy of victim selection. Section 6 shows a preliminary result for the suggested LMK filter framework and the module. The remaining sections offer discussion and conclusions.

2 Previous Approaches for Memory Overload Management in Linux

There have been several approaches to handling memory overload in swap-less devices. As described in Section 1, the approaches can be categorized into two types. The type-A approach is to give hints with `oom_score_adj`[1] to the kernel and kill a process in the kernel with the hints. The type-B approach is to kill a process in the user-space after receiving low memory notification from the kernel. In this section, we briefly describe these approaches.

[1]oom_score_adj is used as the adjustment of each process's attractiveness to the OOM killer. The variable is also used as hints in LMK

Category	Status
System/Persistent	Process is system or persistent process
Foreground	Process is in the foreground or related to the foreground application.
Visible	Process is visible or related to visible applications.
Perceptible	Process is not interacting with user but it can be perceptible to user.
Heavy	Process has cantSaveState flag in its manifest file.
Backup	Process has backup agent currently work on.
Service A/B	Process hosts service.
Home	Process is home application (like Android launcher)
Previous	Process was foreground application at previous.
Hidden	Process is in the background with no above condition.
Empty	Process has no activity and no service.

Table 1: Process Categorization of Android v4.3

2.1 Type-A Approach - Android Low Memory Killer

Android LMK is one of the type-A approaches. The Android platform gives hints to the Android LMK with `oom_score_adj`, and Android LMK selects a victim based on the given `oom_score_adj`. When the operating system triggers the Android LMK, Android LMK determines the minimum `oom_score_adj` of a process to be killed according to six-level memory thresholds and six-level `oom_score_adj` thresholds defined by the Android platform. Android LMK selects processes as victim candidates when they have an `oom_score_adj` higher than the minimum `oom_score_adj`, and it kills the process with the highest `oom_score_adj`. If there are several processes with the highest `oom_score_adj`, it selects the process with the largest memory as the final victim.

To give hints to the Android LMK, the Android platform categorizes processes according to the status of process components. Table 2.1 shows the process categorization of Android V4.3. Based on the categorization, the Android activity manager assigns the proper value to the `oom_score_adj` of each process. Thus, six thresholds of the Android LMK are closely related to the Android process categorization, because the Android LMK tries to kill an application in a specific categorization at a specific memory threshold.

Hidden or empty process categories in the low priority group must be prioritized. Android internally manages a processes list based on LRU according to the launched or resumed time of applications. For the hidden or empty process category, Android assigns process priority based on the LRU. That is, if a process has been more recently used, Android assigns a high priority value to that process. As a result, Android LMK is likely to kill the least recently used process in the hidden or empty process category.

Android LMK has been used for a long time in several embedded devices. However, it is not easy to apply a new victim selection policy because the activity manager of the Android or Android LMK must be modified.

2.2 Type-B Approach - Memory Pressure Notification

Several developers have tried to notify the user-space of memory pressure. The Nokia out-of-memory notifier is one of the early attempts. It attaches to the Linux security module (LSM). [10] Whenever the kernel checks that a process has enough memory to allocate a new virtual mapping, the kernel triggers the low memory handler. At that time, the module decides to send a notification to the user-space through the uevent if the number of available pages is lower than a user-defined threshold. The module sends a level1 notification or a level2 notification based on the user-defined thresholds. However, this method has been hard to use as a generic notification layer for a type-B approach because it only consider a user-space allocation request.

The mem_notify_patch is one of the generic memory pressure notifications. [6] The mem_notify patch sends a low memory notification to applications like the `SIGDANGER` signal of AIX. Internally, it was integrated into the page reclaim routine of the Linux kernel. It triggers a low memory notification when an anonymous page tries to move to the inactive list. If an application polls the "/dev/mem_notify" device node, the application can get the notification signal. The concept of the approach has led to other approaches like the Linux VM pressure notifications [12] and the mem-pressure control group [15]. Such approaches have improved the memory pressure notification.

In type-B approach, user-space programs have the responsibility of handling low memory notification with a policy because the memory pressure notification is the backend of the type-B approach. Thus, the type-B approach expects that the user-space programs release their cache immediately or kill themselves. However, the expectation is somewhat optimistic because all user-space programs may ignore the notification, or user-space programs may handle the notification belatedly. As a result, the system is likely to fall into out-of-memory in such situations. Thus, kernel/user-space mixed solutions, as shown Figure 1-(c), have been developed to improve the limitation of the type-B approach. The Tizen lowmem notifier is one of those hybrid approaches. [16] The Tizen lowmem notifier provides the low memory notification mechanism, and the safeguard killer kills an application based on the `oom_score_adj` given by a user-space daemon when the available memory of the system is very low. Thus, the safeguard killer prevents a system from falling into out-of-memory even when the user-space programs neglect to handle a low memory notification. However, the structure of the low memory handler is quite complicated.

2.3 Type-B Approach - Userland Low Memory Killer Daemon (ulmkd)

Ulmkd is one of frontend solutions of the type-B approach using generic memory pressure notification. The default notification backend of ulmkd is a low memory notification layer on top of cgroups. After receiving a memory notification from the kernel, ulmkd behaves the same way as the Android LMK driver by reading the `oom_score_adj` of each process from the proc file system. Thus, ulmkd needs to read the memory usage information of the system/process from the kernel. As a result, ulmkd can cause a lot of unnecessary system call in a low memory situation. In addition, the process page of ulmkd should not be reclaimed, to prevent unintended memory allocation by page remapping. Although the author of ulmkd tries to solve the issues by using approaches like locking of the process page, and using task-list management of specific user-space platform components, ulmkd still has issues to resolve. [11]

Although it requires lots of stability testing before applying to commercial devices, due to radical issues of user-space LMK, it is quite attractive because it make it easier to change the victim selection policy than the type-A approach. However, ulmkd does not provide a framework to apply the new victim selection policy dynamically or to apply multiple victim selection policies.

To the best of our knowledge, there are no existing solutions for extending the victim selection policy. A similar issue for the OOM killer was discussed at the 2013 LSF/MM summit and an idea to apply a new victim selection policy was suggested [3]. The idea is to create a framework similar to packet filters in the network stack. In this paper, we will present such a framework for dynamically extending victim selection policy.

3 Previous Studies for Enhancing Victim Selection Policy

Although mechanisms for handling the low memory situation is major topic in the industry, not much work has been done for enhancing victim selection policy. However, there are several benefits to enhancing the victim selection policy. In this section, we introduce such studies and benefits.

3.1 Considering Expected Delay

Yi-Fan Chung et al. utilized the concept of *expected delay penalty* to enhance victim selection [2]. *Expected delay penalty* was calculated by multiplying application launch probability by application launching time. If an application has a high *expected delay penalty*, the application is kept in the background. As a result, frequently used applications with long launch times are kept in the background instead of being killed.

3.2 Considering Undesirable Applications

Yi-Fan Chung et al. also utilized the concept of *expected power saving* to enhance victim selection. [2] After they calculated the power consumption of a background application, they calculated *expected power saving* by multiplying the application launch probability by the background power consumption of each background application. If an application has low *expected power saving*, it is kept in the background. Thus, less frequently used applications with high power consumption in the background are selected as victims.

Mingyuan Xia et al. studied how memory-leaking applications can easily cripple background application management with victim selection based on LRU. [13] They noted that normal applications lost the opportunity to be cached in the background when memory-leaking applications were kept in the background. They implemented a light weight memory-leak detector and they modified the Android process prioritization policy. If the detector detects a suspected memory-leaking application, the process is set to the lowest priority. Thus, Android LMK kills the suspected memory-leaking application.

3.3 Considering Personalized Factors

Tingxin Yan et al. predicted applications to be prelaunched by investigating application usage behaviour. [14] They investigated three application usage patterns: "follow-trigger", "location clustering", and "temporal burst". The application usage behaviours were used to predict which applications were likely to be used in the near futures, and the system assigned high priority to such applications. If applications with high priority did not exist in the background, the applications were prelaunched. Likewise, if applications with low priority existed in the background, the applications were selected as victims.

Predicting applications to be used in the near future can enhance victim selection policy. There have been several studies focused on predicting applications that will be used in the near future. Choonsung Shin et al. found that it was effective to predict applications from the last application used, and Cell ID and time of day. [9]. Xun Zou et al. showed a solution for predicting applications from the latest application used based on the Markov model. [17]

If a system provides a mechanism to easily apply the described victim selection policies, it would improve the user experience and system performance. In this paper, we present the LMK filter framework to apply such victim selection policies, and we show the system improvement by applying a new victim selection policy based on a prediction mechanism.

4 The Policy-extendable LMK Filter Framework

In this section, we present a policy-extendable LMK filter framework to extend new policies to an existing

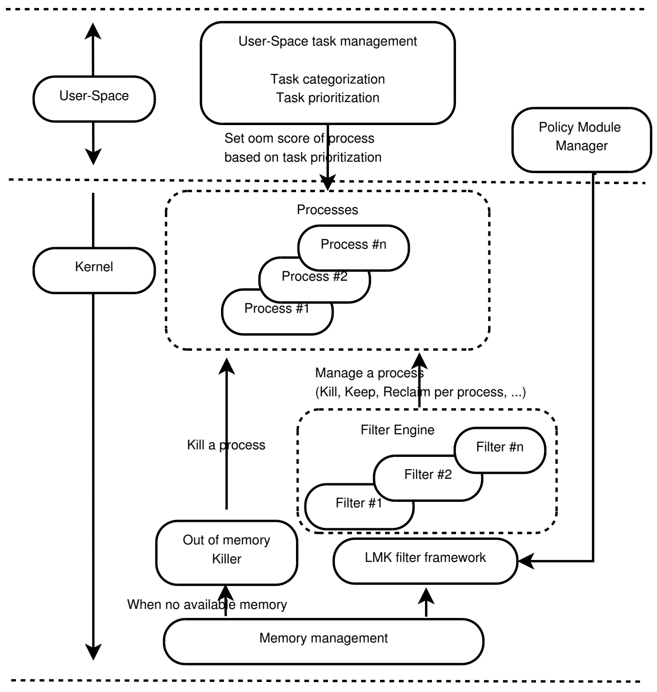


Figure 2: Architecture of LMK filter framework

LMK for a specific purpose or for general improvement. With the LMK filter framework, a module with a new victim selection policy can be applied at any time without modification. The purpose of the LMK filter framework is to provide following functionality.

- Adjust the victim selection policy of the LMK engine by adding/removing/changing policy modules;
- Support multiple policy modules effectively.

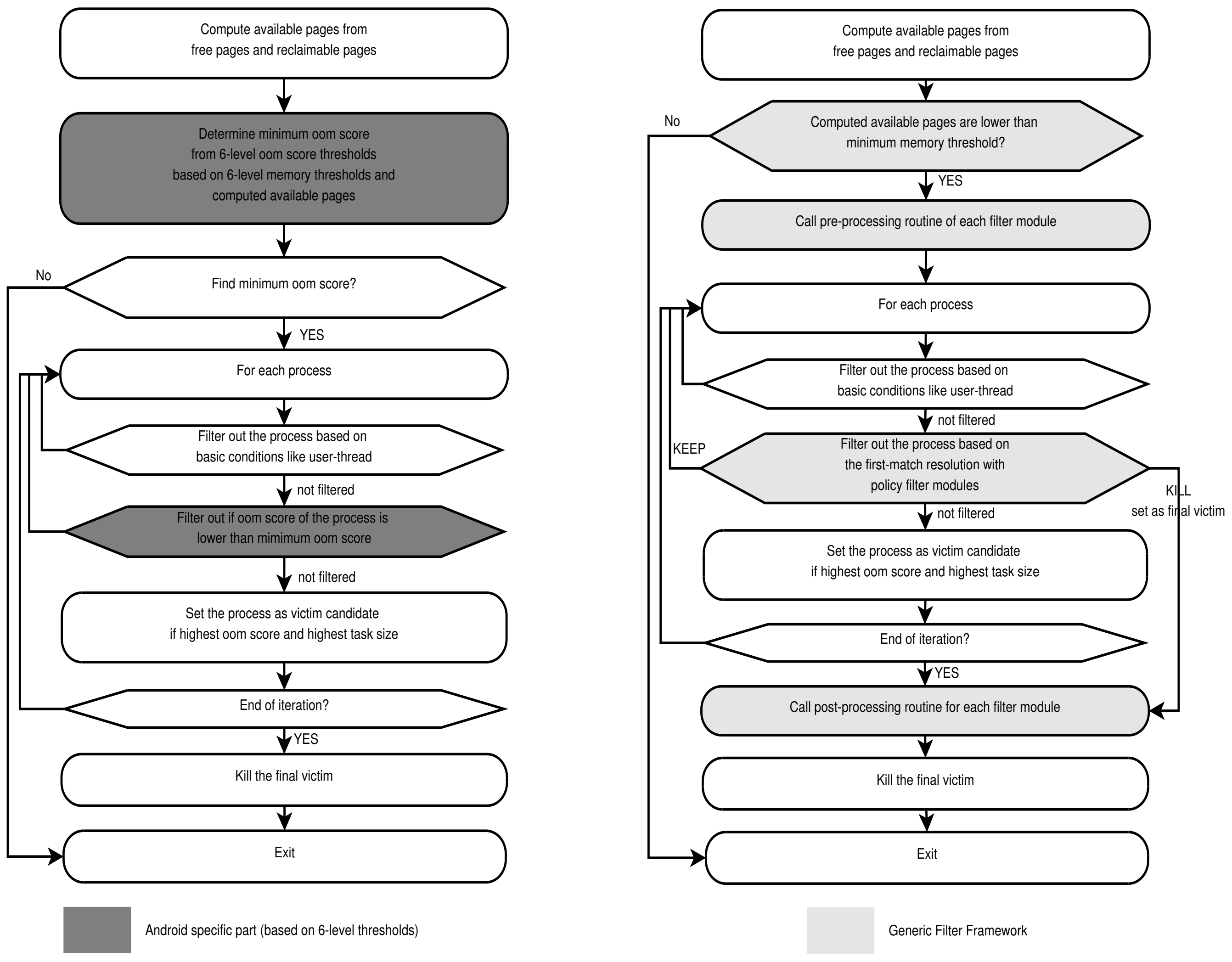
To adjust the policy of LMK in runtime, a policy is consisted of an independent device module. The policy of LMK is adjusted by managing the device module with a policy. To support multiple policy modules, the LMK filter framework implements a first-match resolution mechanism based on the order of policies. With the first-match resolution, the LMK filter framework minimizes a policy conflict and a policy redundancy by the installed several policies. In our implementation, we have termed a policy module and a first-match resolution engine to a filter module and a filter engine.

Figure 2 shows the architecture of the low memory handler with the LMK filter framework. The LMK filter framework and filters replace the LMK driver of the type-A approach. To create a policy-extendable LMK filter framework, we modified the Android LMK because the Android LMK has been used popularly. The type-B approach is more suitable for the LMK filter framework due to its flexibility. However, we chose the type-A approach because the type-B approach has still issues to solve. Although we implemented the LMK filter framework to the type-A approach, it is not difficult to apply to the type-B approach.

4.1 Re-factoring Android LMK

To create a generic LMK filter framework without changing the behaviour of the Android LMK, we analyzed the flow of the Android LMK. Figure 3-(a) shows the detail flow of Android LMK. Android LMK has a generic part for checking available memory, and for selecting an application as victim based on `oom_score_adj`. In addition, Android LMK has a specific part for filtering applications based on the six-level thresholds. Thus, we replaced the specific part with the filter engine routines of the LMK filter framework. Figure 3-(b) shows the re-factored Android LMK.

We replaced the "determine minimum `oom_score_adj` based on six-level thresholds" stage with a generic preprocessing stage for each filter module, and we replaced the "filter out processes based on minimum `oom_score_adj`" stage with a generic filtering stage for each filter module. Finally, we placed a generic postprocessing stage for each filter module after iterating processes. Thus, the sequence of the modified Android LMK is the following. The modified LMK checks the available memory including reclaimable memory, and checks the minimal memory threshold to fall routines for killing a process. After that, the LMK calls preprocessing routines for each filter module to prepare each filter module. After the filtering out processes performed by the filtering routines of each filter module, a process with the highest `oom_score_adj` or a process decided by a filter module is selected as victim, Finally, the LMK kills the victim after calling post-processing routines for each filter module. If the LMK filter framework does not have any filters, the LMK kills the process with the highest `oom_score_adj`. That is, the default behaviour of the LMK filter framework is to kill a process by generic decision based on `oom_score_adj`. We discuss the default behaviour further in Section 7


Compute available pages from free pages and reclaimable pages
Determine minimum oom score from 6-level oom score thresholds based on 6-level memory thresholds and computed available pages
Find minimum oom score?
No
YES
For each process
Filter out the process based on basic conditions like user-thread
not filtered
Filter out if oom score of the process is lower than mimimum oom score
not filtered
Set the process as victim candidate if highest oom score and highest task size
End of iteration?
YES
Kill the final victim
Exit
Android specific part (based on 6-level thresholds)
Compute available pages from free pages and reclaimable pages
Computed available pages are lower than minimum memory threshold?
No
YES
Call pre-processing routine of each filter module
For each process
Filter out the process based on basic conditions like user-thread
not filtered
KEEP
Filter out the process based on the first-match resolution with policy filter modules
KILL set as final victim
not filtered
Set the process as victim candidate if highest oom score and highest task size
End of iteration?
YES
Call post-processing routine for each filter module
Kill the final victim
Exit
Generic Filter Framework

(a) Original Android LMK

(b) Our Approach - Generic LMK Filter Framework

Figure 3: Comparison between native LMK and our approach

4.2 The LMK filter framework

The LMK filter framework consists of two parts: the filter chain manager and the filter engine. The filter chain manager manages the list of filter modules and the lifecycle of filter modules. The manager supports to add a new filter module to the LMK filter engine, and to remove an existing filter module from the LMK filter engine. In addition, the manager supports to change the order of filter modules for the first-match resolution.

The filter engine is the core of the LMK filter framework. The filter engine operates the first-match resolution in the low memory situation and exposes filtering interfaces for filter modules. Figure 4 shows the first-match resolution flow for a process. If a filter module decides that a process should be preserved in the background in the filtering stage, that process is not killed and the traversing filter chain for the process is terminated. Likewise, if a filter module decides that a process should be killed, the filter engine sets the process as a victim and the traversing filter chain of the process is terminated. If a filter module does not make any decision for a process, the filter engine calls the filtering routine of the next filter module in the filter chain. This means that filter modules have a priority based on their position in the filter chain.

The filter engine provides an interface for a filter module. With the interface, each filter module is required to register three routines: a pre-processing routine, post-processing routine, and filtering routine. As shown in Figure 3-(b), the pre-processing and post-process rou-

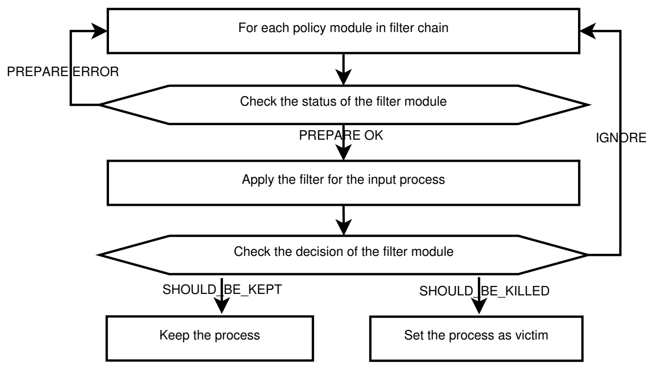

For each policy module in filter chain
PREPARE ERROR
Check the status of the filter module
PREPARE OK
IGNORE
Apply the filter for the input process
Check the decision of the filter module
SHOULD_BE_KEPT
SHOULD_BE_KILLED
Keep the process
Set the process as victim

Figure 4: The first-match resolution of filter engine

tines are called once in a low memory handling flow, and the filtering routine is called for every process, whenever the filter engine iterates processes.

In a pre-processing routine, a filter module must run preparation to filter a process like setting a filter-owned threshold. A module should return PREPARE_DONE or PREPARE_ERROR in the routine. If a module returns PREPARE_ERROR for any reason, the filter engine excludes the filter module in the low memory handling flow. In a filtering routine, a filter module can filter a process based on the filter module's specific victim selection policy. In a filtering routine, a filter module can return one of three decisions for each process: IGNORE, SHOULD_BE_VICTIM, SHOULD_BE_KEPT. If a filter module returns SHOULD_BE_VICTIM for a process, the process is set as a victim. If a filter module returns SHOULD_BE_KEPT, the process is kept in the background. When a filter module does not make any decision for a process, the filter module can return IGNORE.

4.3 Android Specific Filter Module

To execute the same operation with a native Android LMK, we implemented a filter module for the Android. The module implements the a specific part of Android LMK.

In the pre-processing routine, the module finds the minimum oom_score_adj from the six-level oom_score_adj thresholds defined by a user-space platform after comparing the six-level memory thresholds defined by a user-space platform with the calculated available memory pages. If the module finds the minimum oom_score_adj, the module returns a PREPARE_DONE value. In the filtering routine, the module compares the oom_score_adj of a process to the minimum oom_score_adj decided in the preparation stage. If the oom_score_adj of a process is higher than the minimum oom_score_adj, the module returns IGNORE. Otherwise, the module returns SHOULD_BE_KEPT. Thus, if the oom_score_adj of processes are lower than the minimum oom_score_adj, the processes will be kept in the background. If there are other filters, other filters modules will decide the termination of the ignored processes.

5 The Task-Prediction Filter Module

To show a policy-extension with the LMK filter framework, we implemented a new policy module. The victim selection in Android is decided by application categorization and LRU. Therefore, the Android LMK may select an application to be re-used in the near future as a victim because recently used processes may not be reused in the near future. The accuracy of victim selection can be improved by carefully predicting applications what will be reused in the near future. In particular, the last application provides hints to predict an application reused in the near future [9]. To extend the study, we considered the last N-applications. In addition, we considered the memory used by a process to give a penalty for a process using large memory. Thus, we suggest a filter module to keep processes to be reused in the near future based on the last N-application and process memory usage.

The main functionality of the filter module consists of two parts. The first is to maintain an application transition probability matrix and an application usage history, and the other is the filter-preparation function and the filtering function of the LMK filter module provided for keeping the predicted applications in a low memory situation.

5.1 Training prediction matrix

To predict which application would be used next from last N-applications, the transition probability between applications was observed during a training phase. The trained transition probability matrix was used as a Markov chain matrix to predict applications which could be used in the next step. To determine the transition probability between applications, we tracked the foreground application of the Android. To track the foreground application, we modified the write operation of `proc_oom_adjust_operations` in the Linux kernel. Whenever the `oom_score_adj` of a process is changed, the `oom_score_adj` hook function of our filter module is called. The hook function logs the foreground application, which has the foreground `oom_score_adj` of the Android platform. When the hook function recognizes a foreground application, the module inserts the information of the application into a queue which has finite-length to maintain the application usages history, and the module updates the information of a transition probability matrix by relationship.

5.2 Implementation of the filter module

To predict applications to be reused in the near future, we utilized the Markov chain models for link prediction of Ramesh R. Sarukkai.[7] From the model, a probability of the next transition from n-past information can be acquired. Thus, the probability of next application's transition from last N-applications is acquired with the model.

Let M represent the trained application transition probability matrix, and let S(t) represent the state of a foreground application at time t. The following formula derives the transition probability using the Markov chain models for link prediction of Ramesh R. Sarukkai.

$$s(t+1) = \alpha_1 s(t)M + \alpha_2 s(t-1)M^2 + \alpha_3 s(t-2)M^3 + ...$$

In the pre-processing routine, the filter module prepares the transition probability matrix for the formula. In the filtering routine, we generate the final transition probability from the formula.

To filter a process, we considered an additional factor – the memory usage of a process. Although the transition probability of a process is higher than others, if the memory usage of the process is higher than others, it is possible that keeping the process will cause a decrease in the total number of applications kept in the background. That is, when the prediction is incorrect, keeping a process with large memory is likely to produce side-effects. Thus, to reduce such side-effect, the memory usage of a process was also considered with the transition probability.

Based on the transition probability derived from the formula and the memory usage of a process, the filter module computes a transition score. If an application has a high transition probability and low memory usage, the application has a high transition score. Otherwise, if an application has a low transition probability and high memory usage, the application has a low transition score.

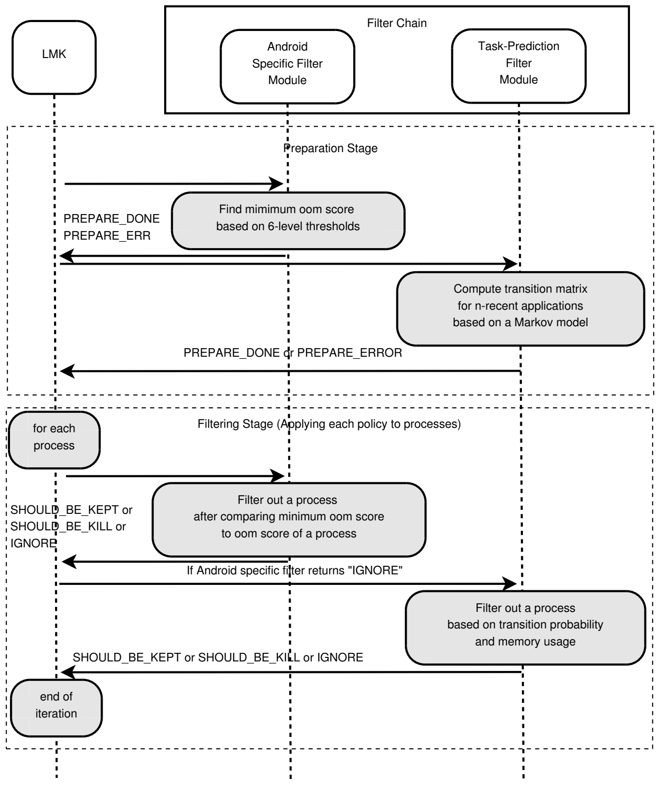

LMK
Filter Chain
Android Specific Filter Module
Task-Prediction Filter Module
Preparation Stage
Find mimimum oom score based on 6-level thresholds
PREPARE_DONE
PREPARE_ERR
Compute transition matrix for n-recent applications based on a Markov model
PREPARE_DONE or PREPARE_ERROR
for each process
Filtering Stage (Applying each policy to processes)
Filter out a process after comparing minimum oom score to oom score of a process
SHOULD_BE_KEPT or
SHOULD_BE_KILL or
IGNORE
If Android specific filter returns "IGNORE"
Filter out a process based on transition probability and memory usage
SHOULD_BE_KEPT or SHOULD_BE_KILL or IGNORE
end of iteration

Figure 5: LMK filter flows with the task-prediction filter module

As a result, the filtering function of the filter module keeps applications with a high transition probability and low memory usage, after computing the transition score. If a process has a high transition score, the module returns `SHOULD_BE_KEPT` for the process. Otherwise, the module returns `IGNORE`. Thus, the filter module tries to keep applications that will be reused in the near future from recent applications with minimal side effects.

The task-prediction filter module is inserted as the second filter module in Android device. Thus, the filter module enhances the LRU-based victim selection policy of Android without changing a victim selection policy by application categorization. Figure 5 shows the sequence of the LMK decision after inserting the task-prediction filter module with an Android specific filter module. The filter module decides the life of a process when the Android specific filter module does not make a decision about the life of the process.

6 Evaluation

In this section, we evaluate the effectiveness of the LMK filter framework and the task-prediction filter module. We implemented the LMK filter framework and the filter module on a Galaxy S4 and Odroid-XU using the Linux 3.4.5 kernel.

Android platform supports process-limit feature to limit the number of background applications. Based on the number of process limitation, Android platform kills an application in background application cache. Thus, if a device has enough memory, the LMK is rarely triggered because Android platform is likely to manage background applications based on process limit before falling into low memory situation. Unfortunately, two devices in our experiments equips enough memory for stacking the default number of process-limit. Thus, the LMK is rarely happen in the devices. Thus, to evaluate our algorithm, we adjusted the amount of memory of the two devices instead of increasing the process limit of Android. We discuss the process limit further in Section 7.

To evaluate our scheme, we created an evaluation framework based on real usage data in a smartphone. With the evaluation framework, we show that the number of application terminations is reduced. In a specific application usage sequence, the decrease of the number of application terminations means the increase of the hit rate for the background application reuse. In addition, it means the decrease of the data loss probability by the LMK forced termination.

6.1 Evaluation Framework

We created an evaluation framework to run applications automatically according to a given input sequence of applications. The framework launches an application in order from the given input applications sequence. To launch an application, the framework executes an activity manager tool through the shell command of android debug bridge (adb). The framework reads the input sequence of application and launches the next application every 10 seconds. At the same time it is performing the application launch, the evaluation framework monitors the number of application terminations provided by the LMK filter framework.

We referenced the usage sequence of applications collected from the Rice LiveLab user study to evaluate our scheme [8, 4]. We generated the input sequence by finding corresponding Android applications in Google Play after extracting a part of the original sequence. Table-2 shows information about the input sequences of each users.

User	Length of input seq.	Usage duration
A00	162	4 days 10 hours
A01	139	1 day
A03	233	2 days 21 hours
A07	154	6 days 11 hours
A10	218	8 days 8 hours
A11	179	6 days 19 hours
A12	159	3 days 9 hours

Table 2: Extracted input sequences

6.2 Estimation

We evaluated our frameworks by using 3-fold cross validation with each user's application usage data. One third of the total sequence was used for training set. We also conducted online training when a test set is tested. The task-prediction filter module predicted next applications from 1-recent applications history to 4-recent applications history. Each user's application usage data were tested five times, and we took the average from the results. Figure 6 shows the number of application terminations in both native LMK and the LMK filter framework with the task-prediction filter module when the filter module predicts the next application from 4-recent applications history. The result was normalized to the number of application terminations in native LMK.

The LMK filter framework with the task-prediction filter module was improved by average 14% compared with the native LMK. In addition, our proposed scheme gave better results for most users. In case of specific user a10, the number of application terminations was improved by 26%. Although we did not experiment with all real usage data, the preliminary result proves that an enhanced application replacement algorithm can be created easily by the LMK filter framework without modifying the entire process prioritization. In addition, in case of specific user a11, the number of application terminations was increased. It shows that a specific policy is not easy to fit to all users because the most appropriate policy for each user differ. The LMK filter framework can solve such problems by applying other policy modules for each user.

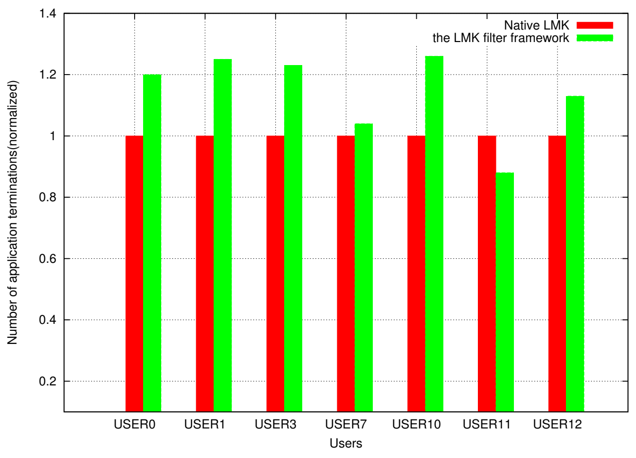


Figure 6: Number of application termination in both native LMK and LMK filter framework with the task-prediction filter module

Routine	N-recent applications	Overheads
Pre-Processing	1	31549 ns
	2	74020 ns
	3	109090 ns
	4	146699 ns
Filtering	1	3371 ns
	2	3522 ns
	3	3632 ns
	4	3411 ns

Table 3: Average overheads according to n-recent applications

6.3 Overheads

We observed the overheads of the LMK filter frameworks and filter modules. The overheads was measured in the pre-processing stage and the filtering stage. Table 6.3 shows the overheads. Most overheads was caused by the task-prediction filter module. In the pre-processing stage, the task-prediction filter module multiplies the transition matrix by n times according to n-recent applications history. Thus, the overheads were increased according to n. In the filtering stage, the filter module computes transition scores. The table shows computation the overheads of computing transition score about a process. Thus, the total overheads can be obtained by multiplying the number of iterated processes in the LMK filter framework. In our experiments, it was not significant overheads because the average number of iterated processes was about 10.

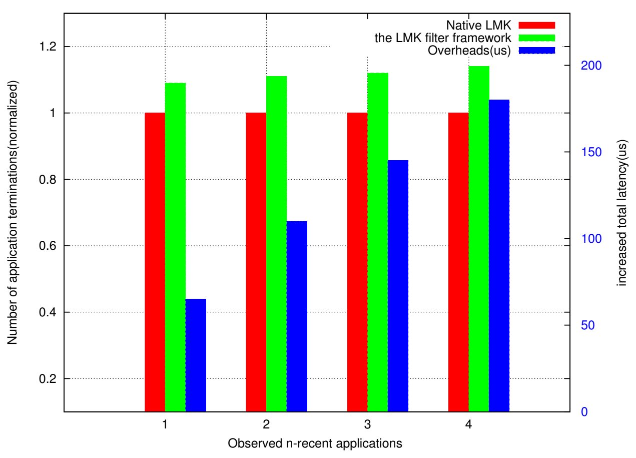


Figure 7: Average number of application terminations and overheads according to n-recent applications

6.4 Accuracy of the Task-Prediction Filter Module

The task-prediction filter module enhances the accuracy of victim selection by predicting the next application from the n-recent applications history. If many n-recent applications are used, application prediction is more accurate. However, the computation overhead is also increased significantly by matrix multiplications. Figure 7 shows the number of application terminations and overheads according to n-recent applications history. As the number of recent applications used for prediction increased, the number of application terminations was slightly reduced. However, the performance overhead is also increased by matrix multiplication. Thus, the n value should be applied properly according to the degree of urgency when handling a low memory situation.

7 Discussion

7.1 The default behaviour of the LMK Filter Framework

The default behavior of LMK filter framework is to kill a process with the highest `oom_score_adj`, and to kill a process with the largest memory when there are several processes with the highest `oom_score_adj`. However, such behavior may cause a reduction in the total number of applications kept in the background. If many applications with large memory are kept in the background, the total number of applications kept in the background is reduced because the memory capacity of the device is limited. Thus, the OOM killer of the kernel decides the badness of a process based on the percentage of process memory in total memory. The oom_score_adj of a process is used to adjust it. As a result, the behavior of the OOM killer is likely to keep more applications in the background.

For embedded devices, we believe that the oom_score_adj of applications given by a specific rules of user-space, such as application categorization of Android, is more important than the percentage of process memory in total memory because such hints include user experience related factors, such as visibility to user. However, this should be decided carefully after further experiments in several embedded devices with several users. To do that, the default behavior of the LMK filter framework also should be changeable. The works are left as our future work.

7.2 The first-match resolution mechanism

The LMK filter framework implements a first-match resolution mechanism. The mechanism is effective to minimize policy conflicts and to reduce redundant operations by policy redundancy. However, to integrate several policies, the order of policy modules should be decided carefully. For example, suppose that there are three policy modules: a memory-leak based policy module, the task-prediction module, the Android specific module. The memory-leak based policy module selects a memory-leak suspected application as victim, as described Section-3. If the order of the policy module is "the Android specific module – the task-prediction module – the memory-leak based policy module", a memory-leak suspected application can be kept in the background by previous two policy modules. Thus, the order of policy module should considered carefully for desirable integration of several policy modules.

7.3 Other Benefits of the LMK Filter Framework

The specific victim selection policies described in Section-3 were applied by individually modifying the process prioritization routine of the Android platform. However, the policies can be applied by using the LMK filter framework without platform modification. In addition, the policies can be activated at the same time by the LMK filter framework.

Instead of applying a new victim selection policy, a new memory management policy for processes can be applied with the LMK filter framework. For example, per-process page reclamation can be implemented as a filter module. A filter module can reclaim a process's page in the filtering routine after checking a process's reclaimable memory. After per-process reclamation, if the filter module returns SHOULD_BE_KEPT for the process, the process will be kept in the background. Without such a mechanism, although per-process page reclamation is provided to the user-space [5], the process is likely to be killed by LMK in a low memory situation.

7.4 User-space LMK filter framework

Implementing policies in the user-space might allow polices to be applied gracefully because user-space's information can be utilized easily. Meanwhile, the user-space program is unlikely to acquire OS-dependent information like the page-mapping information of the process. Above all, the user-space low memory killer is hard to handle quickly for urgently handling a low memory situation.

Thus, there are advantages and disadvantages with the in-kernel LMK filter framework. We expect that the in-kernel LMK filter framework will be able to apply new polices gracefully if the operating system manages context information, such as the context-aware OS service, to enhance the operating system's behaviour [1]. However, we also believe that the LMK filter framework can be easily applied to the user-space LMK, and the user-space LMK filter framework can apply various victim selection policies dependent on specific applications.

7.5 Victim Selection by Process Limit

A S/W platform for embedded device, such as Android, manages background applications based on the number of process limit. If the number of background processes exceeds a pre-defined number of process limitation, the S/W platform kills an application. Unfortunately, in the type-A approach like Android, the victim selection by process limit is executed in the user-space, and the victim selection by LMK is executed in the kernel. Thus, they try to synchronize the victim selection policy by hints, such as oom_score_adj of a process. In our implementation, we did not consider the synchronization. However, the problem can be easily solved by querying victim selection to LMK filter framework.

8 Conclusion

We presented a policy-extendable LMK filter framework and a filter module to enhance LMK victim selection, and showed the effectiveness of the customized LMK and several benefits provided by the LMK filter framework. Although we implemented it in the kernel, it can be implemented as a user-space daemon. We expect that this policy-extendable LMK filter framework and LMK filter will improve user experience.

Acknowledgments

This research was supported by the International Research & Development Program of the National Research Foundation (NRF) funded by the Ministry of Science, ICT & Future Planning (Grant number: 2013064540).

References

[1] David Chu, Aman Kansal, Jie Liu, and Feng Zhao. Mobile apps: It's time to move up to condos. In *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems*, HotOS'13, pages 16–16, Berkeley, CA, USA, 2011. USENIX Association.

[2] Yi-Fan Chung, Yin-Tsung Lo, and Chung-Ta King. Enhancing user experiences by exploiting energy and launch delay trade-off of mobile multimedia applications. *ACM Trans. Embed. Comput. Syst.*, 12(1s):37:1–37:19, March 2013.

[3] Jonathan Corbet. Lsfmm: Improving the out-of-memory killer. `http://lwn.net/Articles/146861/`, 2013.

[4] Rice Efficient Computing Group. Livelab: Measuring wireless networks and smartphone users in the field. `http://livelab.recg.rice.edu/traces.html/`.

[5] Minchan Kim. mm: Per process reclaim. `http://lwn.net/Articles/544319/`, 2013.

[6] KOSAKI Motohiro. mem_notify v6. `http://lwn.net/Articles/268732/`, 2008.

[7] Ramesh R. Sarukkai. Link prediction and path analysis using markov chains. In *Proceedings of the 9th International World Wide Web Conference on Computer Networks : The International Journal of Computer and Telecommunications Netowrking*, pages 377–386, Amsterdam, The Netherlands, The Netherlands, 2000. North-Holland Publishing Co.

[8] Clayton Shepard, Ahmad Rahmati, Chad Tossell, Lin Zhong, and Phillip Kortum. Livelab: Measuring wireless networks and smartphone users in the field. *SIGMETRICS Perform. Eval. Rev.*, 38(3):15–20, January 2011.

[9] Choonsung Shin, Jin-Hyuk Hong, and Anind K. Dey. Understanding and prediction of mobile application usage for smart phones. In *Proceedings of the 2012 ACM Conference on Ubiquitous Computing*, UbiComp '12, pages 173–182, New York, NY, USA, 2012. ACM.

[10] William McBride Aderson Traynor. Nokia out of memory notifier module. `http://elinux.org/Accurate_Memory_Measurement#Nokia_out-of-memory_notifier_module`, 2006.

[11] Anton Vorontsov. Userspace low memory killer daemon. `https://lwn.net/Articles/511731/`, 2012.

[12] Anton Vorontsov. vmpressure_fd: Linux vm pressure notifications. `http://lwn.net/Articles/524299/`, 2012.

[13] Mingyuan Xia, Wenbo He, Xue Liu, and Jie Liu. Why application errors drain battery easily?: A study of memory leaks in smartphone apps. In *Proceedings of the Workshop on Power-Aware Computing and Systems*, HotPower '13, pages 2:1–2:5, New York, NY, USA, 2013. ACM.

[14] Tingxin Yan, David Chu, Deepak Ganesan, Aman Kansal, and Jie Liu. Fast app launching for mobile devices using predictive user context. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*, MobiSys '12, pages 113–126, New York, NY, USA, 2012. ACM.

[15] Bartlomiej Zolnierkiewicz. The mempressure control group proposal. `http://lwn.net/Articles/531077/`, 2008.

[16] Bartlomiej Zolnierkiewicz. Efficient memory management on mobile devices. In *LinuxCon*, 2013.

[17] Xun Zou, Wangsheng Zhang, Shijian Li, and Gang Pan. Prophet: What app you wish to use next. In *Proceedings of the 2013 ACM Conference on Pervasive and Ubiquitous Computing Adjunct Publication*, UbiComp '13 Adjunct, pages 167–170, New York, NY, USA, 2013. ACM.