An Optimization of Memory Usage Based on the Android Low Memory Management Mechanisms

Linlin Xin[1], Hongjie Fan[2(✉)], and Zhiyi Ma[2]

[1] Advanced Institute of Information Technology, Peking University, Beijing, China
1501220090@pku.edu.cn
[2] School of Electronics Engineering and Computer Science, Peking University, Beijing, China
{hjfan,mazhiyi}@pku.edu.cn

Abstract. When users manipulate low memory Android devices, they frequently encounter the application problem of loading slowly because of limited amount of memory. In particular, more applications installed, problems will occur more frequently. We deeply observe the low memory management mechanism of the Android system and find the system has some shortcomings, such as memory recovery efficiency, unnecessary memory requests. In this paper, we optimize memory usage by improving recovery efficiency, prioritize the use of less memory, prevent the instantaneous increase in memory usage, and reduce unnecessary memory requests. Experimental results in a real environment show that our methods effectively increase the size of free memory, and reduce the phenomenon of application self-startup and association startup.

Keywords: Performance optimization · Low memory management · Auto startup

1 Introduction

Android is a Linux-based, open source operating system which runs on smartphones, tablets, smart TVs, and smart wearable devices [1]. Compared to other embedded systems, Android system has good open source features. Programmers can quickly develop applications without compromise application compatibility, and IT vendors can easily provide feature-specific devices to meet diverse and complex needs.

However, as the number of loading applications increases, the memory requirements make it impossible for running some applications smoothly on low-memory devices. It causes users to load system slowly, especially when more applications installed. Ultimately, the user's experience will be affected under this situation.

As shown in Fig. 1, The effect of system memory is demonstrated in the Google I/O Conference[1]. With the process goes from the top to bottom, the importance is weakened in turn. Basic functions are affected or may even the system is restarted when the system terminates processes such as Home, Service, Perceptible, and Foreground.

[1] https://events.google.com/io2018/.

J. Liu et al. (Eds.): MobiCASE 2020, LNICST 341, pp. 16–36, 2020.
https://doi.org/10.1007/978-3-030-64214-3_2

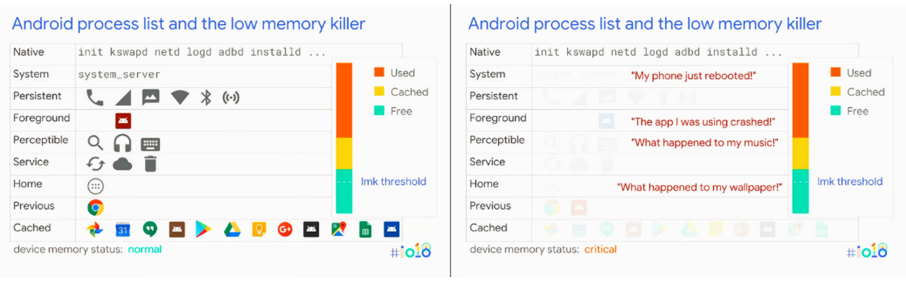


Fig. 1. System memory impact on users.

This phenomenon mainly due to the lack of system memory by two reasons:

1. Android is a multi-tasking system based on Linux system [1]. The corresponding application is executed according to the priority and queue order. Under the situation of low system memory and high CPU load, tasks such as user operations are to be queued for execution and the system is unresponsive.
2. In order to reclaim a portion of the memory, the system has to clean up most of the file cache or even terminate some of the application process. When these applications are used again, the system needs to reallocate the memory to load the file resources, which resulting in a large time expenditure.

As the number of applications increases, the system free memory becomes less. In order to ensure the system has a certain amount of free memory, the system's process will reclaim part of the file cache. Android Low Memory Killer (LMK)[2] is a process monitoring memory and reacting to high memory pressure by killing the least essential process(es) to keep system performing at acceptable levels. Low Memory Killer (LMK) is important whenever available memory of system is below some threshold values [2]. The Low Memory Killer (LMK) is an android specific implementation of OOM Killer (Out Of Memory Killer)[3] mechanism based on Linux. It sets a few adjustment value and minimum free memory pairs while booting OS. LMK is triggered if the amount of available memory is not sufficient. Then LMK kills apps which having a lower adjustment value if the amount of available memory is under a certain threshold. In Android platform, these thresholds are set using min-free values whereas in Linux kernel using watermark levels [3, 4]. Kook et al. [5] proposed a novel selection scheme which runs the OOM killer to terminate arbitrary processes in O(1) time.

The LMK selection process satisfies the following two conditions:

1. Oom_score_adj. Oom_score_adj is calculated by LMK through oom_adj and needs to be greater than the preset min_score_adj value.
2. Largest amount of physical memory. After the process is selected, the LMK sends a SIGKILL signal that cannot be ignored or blocked to terminate the process.

[2] https://android.googlesource.com/platform/system/core/+/master/lmkd/README.md.
[3] https://en.wikipedia.org/wiki/Out_of_memory.

For example, the memory size is configured to 2 GB machines for Android 7.1 system, the low memory threshold rules are shown in Table 1.

Table 1. Rule table of low memory killer in low memory threshold

Free memory is lower than	72 MB	90 MB	108 MB	126 MB	216 MB	315 MB
Corresponding min_score_adj value	0	58	117	176	529	1000
Corresponding oom_adj value	0	1	2	3	9	16

When the system memory is small than 315 MB, LMK would select the appropriate process with oom_adj value high than 16 and terminates it. The retaining process then greatly reduce the startup time when the user is used again. However, in some special cases, LMK has two shortcomings since the low memory threshold and the oom_adj correspondence value are fixed:

First, when the free memory is lower than 216 MB but higher than 126 MB, LMK would terminate the process with the oom_adj value larger than or equal to 9 until the free memory is higher than 216 MB. However, if the system whose oom_adj value is greater than or equal to 9 is all terminated and the free memory is still less than 216 MB, the system will scan all processes again to find a process with an oom_adj value greater than or equal to 9. At this point, most of the time is wasted on the scanning operation, which makes the memory recovery inefficient.

Second, LMK is a passive memory recycling mechanism triggered by the memory threshold level, which consumes a lot of CPU resources. The conditions for terminating the process are only by two factors: the oom_adj value of the process and the size of physical memory occupied by the process. Because the selection process standard is simple, and sometimes even terminates the application that user cares about. In addition, some useless processes are not preferentially reclaimed because they do not reach the preset low memory threshold. For example, when the free memory is just below 216 MB, according to Table 1, the system can terminate the process with the oom_adj value greater than or equal to 9, but there are some inactive background services (B Service processes with oom_adj value of 8). It is not recovered because it is below the preset threshold of 9.

Based on these observations, we deeply study the low memory management mechanism and optimize the system based on shortcomings. We optimize the two aspects of preventing the instantaneous increase of memory usage and reducing the unnecessary memory application. In summary, contributions in this paper are as follows:

1. Modify the Linux kernel layer to improve the recovery efficiency of Low Memory Killer. We introduce the vmpressure formula to measure the memory recovery pressure of the current system. When the system pressure is high, the system selectively terminates the process with a smaller oom_adj value, which helps LMK reduce the number of scans and improve the efficiency of memory recovery.
2. Modify the activity manager service part of application framework layer to preferentially recycle less memory. For service processes that are always working in

the background (for example, B Service), we increase oom_adj value so that LMK can preferentially terminate the least recently used process to free up more memory space.
3. Modify the active service in the application framework layer to prevent the instantaneous growth of memory usage. According to a large number of test results, we adjust the number of parallel startups. Thereby we can reduce the instantaneous memory usage and prevent the pulsed downward trend of the free memory size.
4. Modify the application layer to reduce unnecessary memory requests. We restrict application self-start and association startup for services at application and system level, reducing unnecessary memory requests.

The first two parts are corresponding to supplement and the LMK deficiency optimization. If the system is in a low memory state after optimizing the LMK, we will optimize the latter two parts to avoid the system being in a low memory state.

The rest of the paper is organized as follows: In Sect. 2, we present the related studies and compare those with our study. We describe our platform architecture and methodology in Sect. 3. In Sect. 4, we present our performance evaluations and discussions. Finally, we present the conclusions, and future work in Sect. 5.

2 Related Work

Purkayastha [6] proposed a survey of Android optimization about Android architecture and Android runtime. Hui [7] presents a novel schema for high performance Android systems with using pre-cache on multiple mobile platform. Mario [8] performed an in-depth analysis into whether implementing micro-optimizations can help reduce memory usage. Besides, Mario conducted a survey with 389 open-source developers to understand how they use micro-optimizations to improve the performance of Android apps. Joohyun [9] develop a context-aware application scheduling framework which adaptively unloads and preloads background applications for a joint optimization saving and minimizing the user discomfort from the scheduling.

Android OS [10] has a process terminating function, called Low Memory Killer. The function automatically terminates application processes when the size of available memory become small. There are many approaches proposed to optimize the low memory killer based on app launch patterns. The method proposed in [11] follows the standard framework in Android low memory killer. Different from the standard killer, the approach redefines the importance hierarchy using different options, e.g., the LRU heuristic, app re-forking time, or the highest important level within the two, etc., and then reprioritizes the apps inside a same importance level with metrics derived from the memory consumption of the apps. Without a systematic way, those options are combined in an ad-hoc manner, leading to the experimental result that app re-forking time and derived metrics from app memory consumption do not help on top of the importance level redefined with the LRU heuristic. Their best result is achieved with an option similar to the 2nd baseline in our experiment. Cong [12] proposed the approach to optimize the low memory killer with reinforcement learning. The low memory killer continuously observing various indicators and metrics for memory management, making the process-killing

decisions, and taking app launch latencies as the penalties from the decision-making environment. The low memory killer function automatically terminates application processes when the size of available memory becomes small. Sang-Hoon et al. [13] proposed a novel memory reclamation scheme called *Smart LMK*, which minimizes the impact of the process-level reclamation on user experience. The memory footprint and impending memory demand are estimated from the history of the memory usage keeping an app. *Smart LMK* picks up the least valuable apps and efficiently distinguishes the valuable apps among cached apps and keeps those valuable apps in memory.

Kim et al. [14] proposed heuristic approach to detect the periodic patterns and show that it improves the performance on some specific apps. Researchers in [15] 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. Zhang et al. [16] presents an approach for Android devices which protects certain processes from memory acquisition by process memory relocation. They relocated to the special memory area where the kernel is loaded. Yu et al. [17] propose a two-level software rejuvenation, with the two levels referring to software applications and the OS. Based on this strategy, they construct a Markov regenerative process model to optimize the time required to trigger rejuvenation for Android smartphones. The methods are complementary to ours in terms of refining our simple non-parametric models to predict app launches by embedding the contexts (e.g., the location context) as the model conditions. Some researches monitor how users use the apps and evaluate the interestingness of an app with linear combination of different features [18]. Liang et al. [19] analyze that current memory management algorithms are not working well on Android smartphones and exploiting the tradeoff. Amalfitano et al. [20] present FunesDroid for the automatic detection of memory leaks tied to the Activity Lifecycle in Android apps to detect and characterize these memory leaks. Lee et al. [21] design an estimator to minimize the Frobenius norm of the gain matrices which show excellent performance in environments, where noise information is unknown and in which sudden disturbances are inserted. Ryusuke et al. [22] focus on Android's Generational GC and propose a method for improving its promotion condition. They control promotion based on monitored statistical information that indicates smaller objects tend to die in shorter time. Qing et al. [23] propose MEG, a Memory-efficient Garbled circuit evaluation mechanism, which utilizes batch data transmission and multi-threading to reduce memory consumption.

For improving the real-time capabilities, without loss of original Android functionality and compatibility to existing applications, Igor et al. [24] apply the RT_PREEMPT patch to the Linux kernel, modify essential Android components like the Dalvik virtual machine and introduce a new real-time interface for Android developers. Wook et al. [25] propose a personalized optimization framework for the Android platform which takes advantage of user's application usage patterns in optimizing the performance. Based on this, they implement an app-launching experience optimization technique which tries to minimize user-perceived delays and state loss when a user launches apps. Instead of refining the policy of process-killing, there are other efforts to improve the app launch performance with system level optimization. Some authors attempted to improve performance by modifying AMS. Yang et al. [26] changed the LRU based replacement

policy in the AMS. They suggested a pattern-based replacement algorithm. Ju et al. propose the reclaiming method for mitigating sluggish response based on the MOS Kernel module [27], besides before each app launch demonstrating that the number of kernel function calls is reduced, which make the sluggish response when launching applications. Ahn [28] proposed a system that automatically detects and corrects memory leaks caused by JNI. The system works in detection, correction, and verification. Yang et al. [29] developed an automated tool to acquire the entire content of main memory from a range of Android smartphones and smartwatches. Kassan et al. [30] presents a new self-management of energy based on Proportional Integral Derivative controller (PID) to tune the energy harvesting and Microprocessor Controller Unit (MCU) to control the sensor modes. Maiti et al. [31] present a framework enabling effective and flexible smartphone energy management by cleanly separating energy control mechanisms from management policies.

3 Platform Architecture and Methodology

The overall structure is shown in Fig. 2. First, we use Vmpressure to calculate the memory recovery pressure of the current system by the ratio of Scanned Page Size and Recovered Page Size. According to Vmpressure, we estimate current memory recovery pressure. We terminate a qualifying process and reclaim memory if the system is in low memory. In addition, we provide the priority to recover less used memory called Activity Manager Service. The system updates the oom_adj value of the process according to the importance. If oom_score_adj larger than min_score_adj, we will select this process and turn into the terminate process. Additionally, we reduce the memory usage rate by increasing the maximum number of concurrent starts of the service to alleviate the pressure of memory recovery and give the CPU enough time to recycle. Besides we reduce unnecessary memory requests from the perspective of restricting application self-starting and association startup.

3.1 Improve the Memory Recovery Efficiency

In order to improve the memory recovery efficiency of LMK, we introduce the vmpressure formula to measure the memory recovery pressure of the current system. The main idea of the vmpressure formula is to use the ratio of the unsuccessfully reclaimed memory size to the scanned Page size as a measure of the system memory recovery pressure. The vmpressure formula calculates the current instantaneous memory recovery pressure. The larger the ratio, the system has the greater memory recovery pressure. So when the recovery pressure is large, the process is terminated (the min_score_adj value is lowered). It is worth mentioned that the initial instantaneous pressure index is not very accurate. But over time, this ratio will gradually be averaged and refined to accurately represent the memory recovery pressure of the current system. When the system pressure is large, the system will terminate the process with small oom_adj value and improve the efficiency of memory recovery.

The formula is shown in:

$$Pressure = \frac{Mem_{Scanned} - Mem_{Reclaimed}}{Mem_{Scanned}}$$

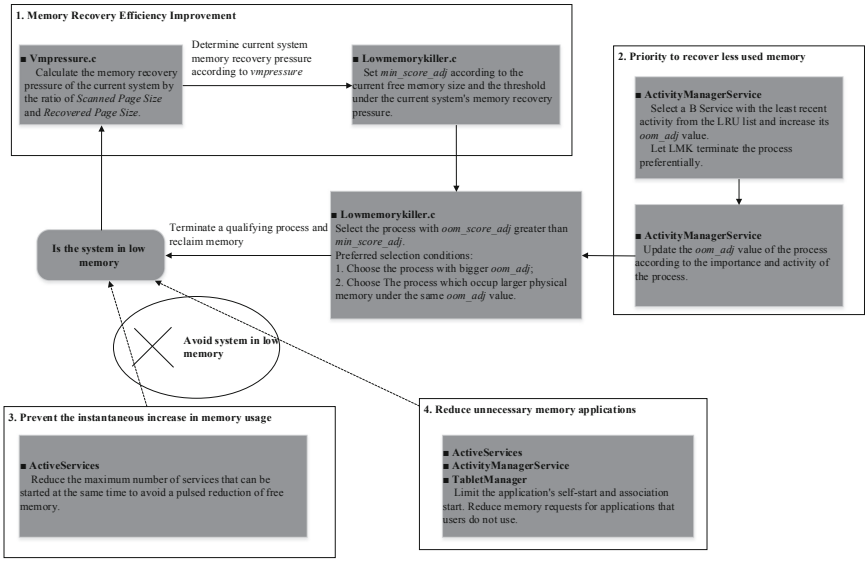


Fig. 2. The workflow of android low memory management.

Where Pressure is the pressure value of the memory recovery, the variable $Mem_{Scanned}$ is the scanned memory size, and the variable $Mem_{Reclaimed}$ is the size of the reclaimable memory. The formula shows that the more memory that is not successfully reclaimed, the memory has the greater recovery pressure of the system.

After the pressure value Pressure is calculated, the reclaimed memory path determines whether to decrease the min_oom_adj value according to the size of the Pressure value and the size of the file cache. The specific rules are shown in Table 2:

Table 2. Optimized low memory killer low memory threshold override rule table.

Free memory is lower than	72 MB	90 MB	108 MB	126 MB	216 MB	315 MB
Original min_score_adj	0	58	117	176	529	1000
Corresponding oom_adj	0	1	2	3	9	16
After the leap, min_score_adj	0	58	58	58	176	176
The corresponding oom_adj after the leap	0	1	1	1	3	3

1. When the pressure value pressure >= 95, and file cache is <= 250 MB, the process is terminated.
2. When the pressure value is 95 > pressure >= 90, and file cache is less than 175 MB, the process is terminated.

3. If the above two conditions are not met, it is considered that the memory recovery pressure is not large, the memory is reclaimed according to the original threshold.

3.2 Prioritize the Use of Less Memory

Low Memory Killer is a passive memory recycling mechanism triggered by the memory threshold level, so it takes up a lot of CPU resources when triggered [12]. The system terminates the process in two conditions: the oom_adj value of the process and the size of physical memory occupied, which are analyzed previously.

In order to prevent the useless process from being reclaimed due to failure to reach the preset low memory threshold, we prevent the system from being in a lower memory state by preferentially recycling fewer processes. The main idea of this method is according to a large number of test verifications, we find the service process working in the background (the B Service process in the code, the oom_adj value is 8), and select the least recently used process according to the order in the LUR list. Since when the process is terminated, only one oom_adj value of the least recently used B Service process is incremented each time, the next B Service process is selected, so that the design is to reclaim less memory while still being used. Keep the process as much as possible. Thus, Low Memory Killer preferentially terminates the least recently used process to free up more memory.

3.3 Prevent the Instantaneous Increase in Memory Usage

The Android system is a multitasking system that allows multiple services to start at the same time. However, when the number of concurrent services is large, the free memory will be pulsed down and resulted a sudden shortage of free memory. As a result, the system will be busy with reclaiming memory work, causing the work of other processes to be delayed. After in-depth research, many applications are secretly launched by the service component without the user's knowledge. Starting the application creates the application's process, the creation process requests a block of memory. Therefore, when the number of concurrent services in the service is large, the instantaneous memory usage will increase, causing a sudden shortage of memory space in the system. In this situation, we reduce the memory usage rate by increasing the maximum number of concurrent starts of the service to alleviate the pressure of memory recovery and give the CPU enough time to recycle.

The phenomenon of service parallel startup is more common when the system sends a system broadcast. For example, after booting, the system will issue a system broadcast with startup completion, such as BOOT_COMPLETED. All event will be started, which will start a service to deal some operations in the background. The default number of service parallel launches for Android system is 8. After extensive testing and analysis, we change the number to 4. If the value is too small, there will be too many services that are queued to start, resulting in a long event delay and affecting the application of system-level services. For example, the alarm application may cause the alarm time to expire, but the alarm service is still in the queue, which brings a bad user experience.

3.4 Reduce Unnecessary Memory Requests

Some applications are used to reduce the first boot time or start by the user without knowing by service or broadcast. It wastes the system memory resources and affects the speed of the system operation. Taking the Lenovo YOGA Table 3 X90F tablet as an example, as shown in Fig. 3, although a new machine does nothing after booting (no application is installed, no network is connected), the current memory state is captured as below after 10 min.

Table 3. Detailed configuration parameters of YOGA flat 3.

Model	YOGA Table 3 X90F
Operating system	Android 6.0.1
System memory	2 GB
Storage	32 GB
CPU model	Intel 2.24 GHz(Quad-Core)
Screen size	10.1 inches
Screen resolution	1280 * 800
Screen ratio	16:10
WiFi function	Support

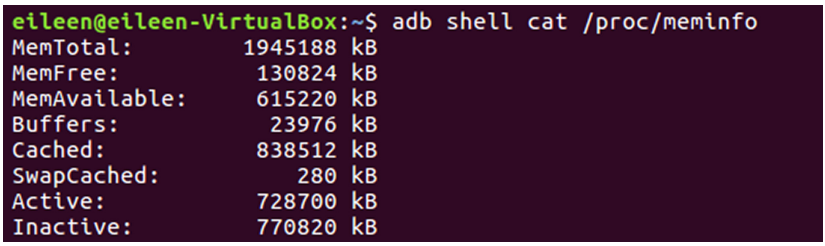


Fig. 3. Memory status after the device is turned on for 10 min.

From the figure, you can find the device with 2 GB memory, minus some memory reserved by the kernel. The actual physical memory (*Mem Total*) of the device is 1945188 KB, about 1.85 GB. The free memory (*Mem Available*) after 10 min of booting is 615220 KB (about 600 MB), and the memory space is already low. If we install some applications that start automatically, the device will be in low memory state after booting, and the system will be busy reclaiming memory. Therefore, it is necessary to reduce unnecessary memory requests. From the perspective of restricting application self-starting and association startup, the system sends two nodes, Broadcast and Start Service, to optimize.

3.4.1 Broadcast Process Optimization

The broadcast is sent to the corresponding *Broadcast Receiver* through the *Activity Manager Service*. First, the *Activity Manager Service* will save all the registered *Broadcast Receiver* to the *Receiver Resolver* object, and parse the incoming data, and find the corresponding *Broadcast Receiver* according to the value of the Intent. Then create a new *Broadcast Record* block with the obtained parameters and add it to the *Broadcast Queue*. The *Activity Manager Service* maintains two *Broadcast Queues* (the foreground *Broadcast Queue* d broadcast queue and the background broadcast queue), which hold all the Broadcast objects that need to be sent. Finally, the *process Next Broadcast* method of the is called by the message mechanism to sequentially process the broadcast in the queue. The transmission and processing is asynchronous.

We add restrictions on sending broadcasts in the *process Next Broadcast* () method of *Broadcast Queue* so that broadcasts that meet the restrictions are not sent. These restrictions correctly intercept broadcast events that are not user-operated, non-system events. After in-depth analysis and extensive testing of Android code, this article adds the following restrictions to the transmission of Broadcast.

1. Broadcast can send if Broadcast belongs to the user-initiated application (that is, the application of the process in the LRU list).
2. Broadcast can be sent if the Intent action of Broadcast is started as a Widget.
3. Broadcast can be sent if the package name of the broadcast is the same as the caller's package name.
4. In other cases, the system checks the policy set by the user and decides whether it can be sent according to the policy.

The above is a restriction policy for sending broadcasts, but most applications are self-starting by receiving broadcasts of Android system events, such as system boot broadcast, device networking broadcast, caller broadcast. These system broadcasts are an integral part of the normal operation of the system. It is no longer possible to make a detailed distinction between the applications here, so it is necessary to make policy restrictions on the startup.

3.4.2 Optimize System Services

Service in the Android system is divided into two categories according to the level: application services and system services. Application services are services defined and implemented in an application. These services are managed in *Active Services.* System services refer to the services necessary for the Android system to work, such as *Activity Manager Service*, *Package Manager Service*, etc., such services are managed in the *Service Manager*. There is a clear difference between the two. Since the middle layer has been standardized in Android design, when the application developer implements a service, it is only necessary to simply implement the server and the proxy. Since this article is to limit the background self-starting of the application, it is only for the Service startup process in *Active Service.* In order to facilitate developers, the Android system encapsulates the Service. So the developer only needs to call the *start Service* or *bind Service* method of the Context. After the call, *Context Impl* will call the *Active Services*

retrieve *Service Locked* method through the corresponding method of *Activity Manager Service* to get the Service record.

The calling process is shown in Fig. 4.

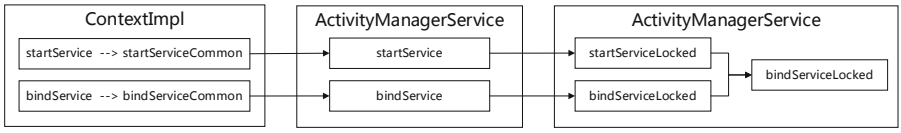


Fig. 4. System service call flow.

Since the two startup methods of Service finally obtain the Service record through the *retrieve Service Locked* method, this article makes policy restrictions when the *retrieve Service Locked* method obtains the Service record. After in-depth analysis and extensive testing of the Android code, the following restrictions are imposed on the startup of the Service.

1. When the *Service Info* of the Service is null, it can be started.
2. When the Service belongs to a user-initiated application (that is, the application of the process in the LRU list), it can be started.
3. When calling the service's process's uid < 10000 and the Service's Intent Action is *Sync Adapter*, check the policy set by the user, and decide whether it can be started according to the policy.
4. When calling the service's process's uid < 10000 and the permission is BIND_JOB_SERVICE, check the policy set by the user and decide whether it can be started according to the policy.
5. When the process of the service is uid < 10000, it can be started, and only the service of the three-party application is restricted.
6. When the uid of the process where the service is located is the same as the uid of the caller, it can be started.
7. When the application package name of the Service is the same as the caller's package name, it can be started.
8. In the remaining cases, check the policy set by the user and decide whether it can be started according to the policy.

The above strategy design is mainly used to limit the self-starting and association startup of the application. Its structure can be divided into the following four parts:

1. Call the interface of the *Tablet Master Service* in the Android system framework. For example, in the *Re-Service Service Locked* method of *Active Services*, the interface of the *Tablet Master Service* is called to determine whether the Service can be started.
2. Launch the interface part of the management to provide the user with an operable interface setting. Users can set which applications can be self-starting and which applications can be launched by association to set up a whitelist.
3. Store the user-set application self-start, association start, and whitelist data into the backend database.

4. The preset configuration file is used to save the list of applications that the user cannot control. There is some system preset applications that do not want to be restricted by users, such as self-starting of applications.

These four parts work together to help users limit the self-starting and association startup of the application. The working flow chart is shown in Fig. 5

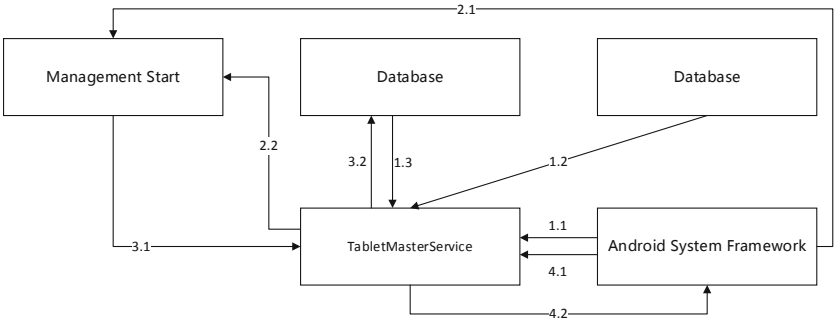


Fig. 5. Workflow chart limiting application self-starting and association startup.

The process in Fig. 5 indicates that when the device is powered on, the *Service Manager* of the Android system starts the *Tablet Master Service*. Then perform the 1.2 operation, 1.2 indicates that the *Tablet Master Service* reads the preset list file, and sets the policy value of the applications that are not controllable by these users to 0. (*Tablet Master Service* divides the application into four categories according to the policy value. The first type of policy value is 0, which is an application that is not controllable by the user, that is, an application that does not participate in the restriction condition. The second policy value is 1, which is set by the user in the application interface. The application is not allowed to start. The third policy value is 2, which is the application that the user is set to allow to start in the application interface. The fourth policy value is 3, which is the application that the user sets to the whitelist in the application interface. Execute startup restrictions for applications in the whitelist). Then perform the 1.3 operation, 1.3 means that the *Tablet Master Service* reads the user settings from the database, and saves the application's package name and policy value in the form of *Hash Map* key-value pairs, and transmits them to the application interface part through the interface.

When the user clicks on the app, the user is listed with a list of all apps that can be restricted, and a switch button is provided for the user to personalize. When the user clicks on the application, the 2.1 operation is performed, and 2.1 indicates that the application obtains a list of all installed applications from the system's *Package Manager*. Then, the 2.2 operation is performed. 2.2 indicates that the application obtains the policy values of the self-starting and association startup of each application from the interface in the *Tablet Master Service*, and merges the list with the 2.1 part into a list, and correctly displays the status of each application on the interface.

When the user modifies the management policy value through the interface, the 3.1 operation is performed. 3.1 indicates that the application invokes the *Tablet Master Service* interface to modify the corresponding policy value. Then perform the 3.2 operation, 3.2 indicates that the *Tablet Master Service* saves the policy value.

When the system sends a broadcast or starts the service, the 4.1 operation is performed. 4.1 indicates that the *process Next Broadcast* () method of the *Broadcast Queue* calls the *check Intent Auto Start* interface of the *Table Master Service* to determine whether the broadcast satisfies the restriction condition and can be sent. Or the *refresh Service Locked*() method of *Active Services* calls the *check Service Auto Start* interface of the *Tablet Master Service* to determine whether the Service satisfies the restriction condition. 4.2 Operation indicates that the *Tablet Master Service* notifies the system of the result, and the system sends or does not send Broadcast, start or not start the Service according to the result.

The above is the workflow of the application. Each module has a clear division of labor and work together to effectively reduce the self-starting and associated startup of the application.

4 Results and Discussion

4.1 Experimental Setup

We run different apps on a mobile device with Intel quad-core, 2 GB memory for performance test and result verification. Detailed parameter configuration information of device is shown in Table 3.

The version of the device was fairly smooth before being optimized. However, after installing 30 mainstream applications, system is often jamming during use. We prepare two YOGA Table 3 X90F tablet devices with the same hardware parameters, one for the native Lenovo system ROM: YT3_X90F_10_row_wifi_20180202, labeled "T-original"; the other is the optimized ROM, labeled "T-optimized". 80 applications are installed on the two tablets. The application list is shown in Fig. 6.

From the perspective of user operation, the system response speed is the main criterion for measuring system performance, such as the time when the application interface switches display, the time when the application is launched after clicking the application icon. From a system internal point of view, memory usage is the primary measure of system performance.

We compare the data of two devices before and after optimization to test whether the solution can optimize system performance and optimization in three ways:

1. Calculate the state of device free memory.
2. Filter analysis and statistics intercepts.
3. Count the interface response speed.

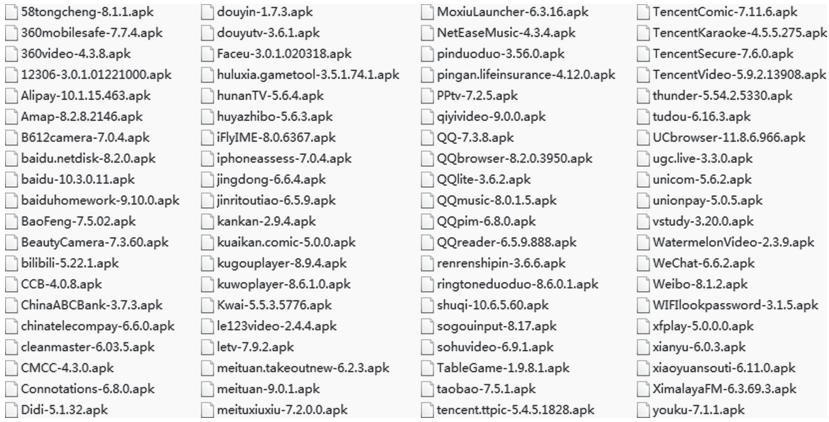


Fig. 6. Application list.

4.2 Experimental Results

4.2.1 System Free Memory

By monitoring the free memory, first we test the solution to improve memory recovery efficiency and prioritize the use of less memory methods to avoid long-term low memory. We use the free memory in the device process as a measure to improve memory reclamation efficiency and prioritize the use of less memory. During the test, we connect the device to the computer using the USB cable, and execute the following command on the computer to check the current memory status of the system:

We conduct two devices (T-original device and T-optimized device) as follows:

(1) After powering on, let it stand for 10 min. Then we check the current memory status of the system, and record the value of "*Mem Available*" (which is the size of free memory).
(2) Open the app list and tap the app's icon. After the system displays the application interface, record the value of "*Mem Available*".
(3) Press the Home button to return to the desktop.
(4) According to the application list, we cycle step 2 and 3 and use the "*Mem Available*" value after the first 15 applications to compare the results of the two devices before and after optimization.

Results are shown in Table 4.

Table 4. Comparison of average free memory.

Operation	Free memory on "T-original" device (Before Optimization)	Free memory on "T-optimized" device (After Optimization)
Power on and leave for 10 minutes	363 MB	615 MB
After using "Connotations"	343 MB	519 MB
After using "Watermelon Video"	237 MB	482 MB
After using "China Mobile"	274 MB	393 MB
After using "Alipay"	229 MB	383 MB
After using "YouKu"	210 MB	314 MB
After using "iFly IME"	207 MB	309 MB
After using "Xiao Yuan Sou Ti"	211 MB	273 MB
After using "Xian Yu"	128 MB	270 MB
After using "XimalayaFM"	125 MB	484 MB
After using "WeChat"	161 MB	387 MB
After using "Weibo"	199 MB (Application Not Responding)	343 MB
After using "NetEase Music"	67 MB (System Crash)	313 MB
After using "Table Game"	222 MB	342 MB
After using "Tencent Video"	183 MB	397 MB
After using "Tencent Comic"	149 MB	338 MB
The Average of Free Memory	**207 MB**	**385 MB**

As shown in Table 4, the memory of T-original of power-on reset is 363 MB after 10 min, which is slightly higher than the operating threshold of 315 MB for Low Memory Killer. After using 2 applications, the free memory is lower than Low Memory Killer's working threshold. As the number of applications increases, the memory resources are in short supply, and the minimum is 67 MB, which may cause crash in the system process. The T-optimized average free memory is 385 MB, and the lowest memory is 270 MB. It is still within the primary threshold range of the Low Memory Killer. At this time, the memory recovery pressure is not too large.

In addition, we view the memory usage of the system by setting the select memory page. After the device is turned on after standing for 24 h, we can analyze the statistics within 3 h of the memory page.

(1) By monitoring the free memory, test whether can improve the memory recovery efficiency. Results are shown in Fig. 7:

We observe the device before optimization for 10 min after the boot is still low, 363 MB, slightly higher than the LMK operating threshold of 315 MB after use 2 applications. The free memory is lower than the working threshold of LMK. As the number of applications used increases, the memory resources are in short supply and the minimum is 67 MB, which causes the system to restart. The optimized device has an average free memory of 385 MB and a minimum memory of 270 MB, which is still in the primary threshold of LMK setting.

(2) By monitoring the memory consumption of the device boot process (because when booting the device needs to start more services, so we need to optimize the number

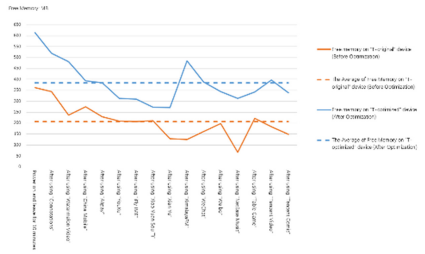


Fig. 7. Comparison of free memory before and after optimization.

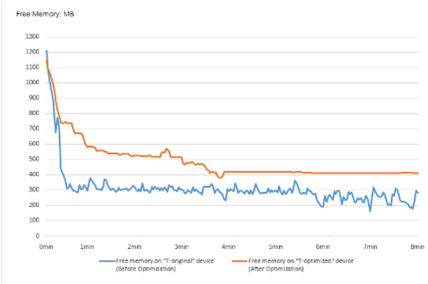


Fig. 8. Variation of free memory during boot process.

of parallel boot of the boot process service), we test prevention of instantaneous growth of memory usage. Results are shown in Fig. 8.

We observe that before optimization ("T-original" device) within 0.5 min of booting, the free memory dropped from about 1200 MB to 300 MB, indicating that the device's instantaneous memory usage is very high. The optimized device ("T-optimized" device) slowly dropped from around 1150 MB to 400 MB within 4 min of booting. It proves that the instantaneous growth of memory usage prevention effectively avoids the pulse growth of device memory and alleviate system CPU pressure.

4.2.2 Self-start and Association Startup Application

In order to eliminate the impact of reducing the unnecessary memory application on the optimization, we set the limit of the T-optimized device from the startup and the associated startup to no limit. Then conduct the following experiments for the two devices ("T-original" device and "T-optimized" device):

(1) Restart the device and execute the./meminfo.sh script.
(2) Stand for 8 min after booting up and use the script to count the "*Mem Available*" value within 8 min to compare the results of the two devices before and after optimization.

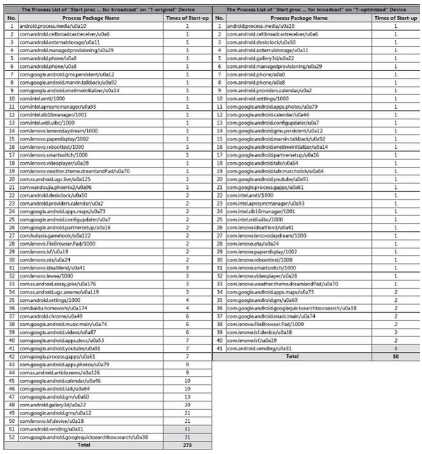
Through the monitoring system log, reducing unnecessary memory requests can effectively intercept application self-starting and association startup. By counting the number of logs without any operation status within 30 min after booting, we show the effectiveness. Results are shown in Table 5.

As shown in Table 5, before optimization ("T-original" device), 52 processes are started by the *Broadcast Receiver* component without any operation, and the cumulative number of startups is 273. 31 processes have been restarted repeatedly. 50 processes are started by the Service component. The cumulative number of startups is 404. Among them, 36 processes have been restarted repeatedly. The most frequently restarted is the "Google GMS" framework, up to 94 times. After analysis, it is found that these processes are repeatedly restarted, causing the device to be in a low memory state. So

Table 5. Process data started by broadcast before and after optimization.

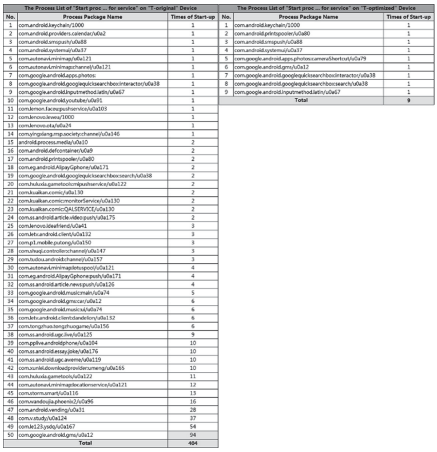
	Before optimization	After optimization
Proc. for broadcast	273 times (52 processes started. 31 of them were restarted, and the most frequent restarts were Google Search applications, up to 31 times.)	50 times (41 processes started, only 8 of them started repeatedly, The number of restarts is up to 3 times.)
Proc. for service	404 times (50 processes are started. 36 of them are restarted repeatedly, and the most frequently restarted are Google GMS applications, up to 94 times.)	9 times (9 processes started, there is no process to restart)

Low Memory Killer needs to continuously terminate the process to release the space, and the terminated process will start again. The optimized device ("T-optimized" device) has 41 processes that are started by the Broadcast Receiver component within 30 min of booting. The cumulative number of startups is only 50, only 8 processes are restarted. The most recent reboot was the "Google Play" app, which was only restarted 3 times. 9 processes are started by the service component and there is no restarted process. The specific situation is shown in Fig. 9 and Fig. 10



The Process List of "Start proc ... for broadcast" on "T-original" Device			The Process List of "Start proc ... for broadcast" on "T-optimized" Device		
No.	Process Package Name	Times of Start-up	No.	Process Package Name	Times of Start-up
1	android.process.media/u0a10	3	1	android.process.media/u0a10	1
2	com.android.cellbroadcastreceiver/u0a6	1	2	com.android.cellbroadcastreceiver/u0a6	1
3	com.android.externalstorage/u0a11	1	3	com.android.deskclock/u0a50	1
4	com.android.managedprovisioning/u0a179	1	4	com.android.externalstorage/u0a11	1
5	com.android.phone/u0a8	1	5	com.android.gallery3d/u0a22	1
6	com.android.phone/u0a8	1	6	com.android.managedprovisioning/u0a29	1
7	com.google.android.gms.persistent/u0a12	1	7	com.android.phone/u0a8	1
8	com.google.android.marvin.talkback/u0a92	1	8	com.android.phone/u0a8	1
9	com.google.android.onetimeinitializer/u0a14	1	9	com.android.providers.calendar/u0a2	1
10	com.android.wmt/1000	1	10	com.android.settings/1000	1
11	com.intel.apinvcmanagers/u0a93	1	11	com.google.android.gsf.apps.photos/u0a179	1
12	com.intel.ui20smanager/1001	1	12	com.google.android.onetimeinitializer/u0a46	1
13	com.intel.uid.ubc/1000	1	13	com.google.android.configupdater/u0a7	1
14	com.lenovo.lenovotipsmm/1000	1	14	com.google.android.gms.persistent/u0a12	1
15	com.lenovo.papershadow/1002	1	15	com.google.android.marvin.talkback/u0a92	1
16	com.lenovo.robocitaxi/2000	1	16	com.google.android.printservice.recommendation/u0a14	1
17	com.lenovo.smartswitch/1000	1	17	com.google.android.setupwizard/u0a56	1
18	com.lenovo.videoplayer/u0a28	1	18	com.google.android.tts/u0a64	1
19	com.lenovo.weather.theme.dreamland.iPad/u0a70	1	19	com.google.android.tts.chromium/u0a66	1
20	com.sun.android.ugc.live/u0a125	1	20	com.google.android.youtube/u0a91	1
21	com.wantoujia.phonetw2/u0a98	1	21	com.google.process.gapps/u0a61	1
22	com.android.deskclock/u0a50	2	22	com.intel.wmt/1000	1
23	com.android.providers.calendar/u0a2	2	23	com.intel.apinvcmanager/u0a93	1
24	com.google.android.apps.maps/u0a73	2	24	com.intel.ubl.fwmanager/1001	1
25	com.google.android.configupdater/u0a7	2	25	com.intel.uid.ubc/1000	1
26	com.google.android.partnersetup/u0a16	2	26	com.lenovo.deskfriends/u0a41	1
27	com.huluda.gamestools/u0a122	2	27	com.lenovo.lancrovoskydream/2009	1
28	com.lenovo.FileBrowserPad/1000	2	28	com.lenovo.otu/u0a2	1
29	com.lenovo.lsf/u0a16	2	29	com.lenovo.papershadowpro/1002	1
30	com.lenovo.ota/u0a24	2	30	com.lenovo.robocitaxi/2000	1
31	com.lenovo.shuttlemp/u0a41	3	31	com.lenovo.smartswitch/1000	1
32	com.lenovo.leos/2000	3	32	com.lenovo.videoplayer/u0a28	1
33	com.sun.android.encalypton/u0a176	3	33	com.lenovo.weather.theme.dreamlandPad/u0a70	1
34	com.sun.android.ugc.zwame/u0a119	3	34	com.google.android.apps.maps/u0a73	3
35	com.android.settings/2000	4	35	com.google.android.tts/u0a66	2
36	com.baidu.homework/u0a174	4	36	com.google.android.googlequicksearchboxsearch/u0a38	2
37	com.android.chromium/u0a49	6	37	com.google.android.music.main/u0a74	2
38	com.google.android.music.main/u0a74	6	38	com.lenovo.FileBrowserPad/1000	2
39	com.google.android.videos/u0a87	6	39	com.lenovo.lsf.device/u0a16	2
40	com.google.android.apps.docs/u0a53	7	40	com.lenovo.lsf/u0a16	2
41	com.google.android.youtube/u0a91	7	41	com.android.vending/u0a31	3
42	com.google.process.gapps/u0a61	7		Total	50
43	com.google.android.apps.photos/u0a79	9			
44	com.sun.android.article.news/u0a126	9			
45	com.google.android.calendar/u0a96	19			
46	com.google.android.tts/u0a66	19			
47	com.google.android.gms/u0a12	13			
48	com.google.android.gsf/u0a16	20			
49	com.google.android.gms/u0a12	21			
50	com.lenovo.lsf.device/u0a16	21			
51	com.android.vending/u0a31	31			
52	com.google.android.googlequicksearchboxsearch/u0a38	31			
	Total	273			

Fig. 9. Optimization of processing data before and after the broadcast started.



The Process List of "Start proc ... for service" on "T-original" Device			The Process List of "Start proc ... for service" on "T-optimized" Device		
No.	Process Package Name	Times of Start-up	No.	Process Package Name	Times of Start-up
1	com.android.keychain/1000	1	1	com.android.keychain/1000	1
2	com.android.providers.calendar/u0a2	1	2	com.android.printspooler/u0a80	1
3	com.android.smspush/u0a88	1	3	com.android.smspush/u0a88	1
4	com.android.systemui/u0a37	1	4	com.android.systemui/u0a37	1
5	com.autonavi.minimap/u0a123	1	5	com.google.android.apps.photos.camerashortcut/u0a79	1
6	com.autonavi.minimapchannel/u0a121	1	6	com.google.android.gms/u0a12	1
7	com.google.android.apps.photos	1	7	com.google.android.googlequicksearchboxinteractor/u0a38	1
8	com.google.android.googlequicksearchboxinteractor/u0a38	1	8	com.google.android.googlequicksearchboxsearch/u0a38	1
9	com.google.android.inputmethod.latin/u0a67	1	9	com.google.android.inputmethod.latin/u0a67	1
10	com.google.android.youtube/u0a91	1		Total	9
11	com.hmon.tacoeguiservice/u0a103	1			
12	com.lenovo.lewea/1000	1			
13	com.lenovo.ota/u0a24	1			
14	com.ydmgdang.mp.societychannel/u0a146	1			
15	android.process.media/u0a10	2			
16	com.android.defcontainer/u0a9	2			
17	com.android.printspooler/u0a80	2			
18	com.eg.android.AlipayGphone/u0a171	2			
19	com.google.android.googlequicksearchboxsearch/u0a38	2			
20	com.huluda.gamestoolspushservice/u0a122	2			
21	com.kuaikan.comic/u0a130	2			
22	com.kuaikan.comic.rnmonitorService/u0a330	2			
23	com.kuaikan.comic.QALSERVICE/u0a330	2			
24	com.sun.android.article.videopush/u0a175	2			
25	com.lenovo.deskfriends/u0a41	3			
26	com.letv.android.client/u0a132	3			
27	com.pl.mobile.putongyu/u0a150	3			
28	com.umeng.soexample.channel/u0a147	3			
29	com.ubsoc.android.channel/u0a157	3			
30	com.autonavi.minimapbiztopic/u0a121	4			
31	com.eg.android.AlipayGphonepush/u0a171	4			
32	com.sun.android.article.newspush/u0a126	4			
33	com.google.android.music.main/u0a74	5			
34	com.google.android.gms.cast/u0a12	6			
35	com.google.android.music.log/u0a74	6			
36	com.letv.android.client.standalone/u0a132	6			
37	com.tongzhan.tongzhuoqame/u0a156	6			
38	com.sun.android.upc.live/u0a125	9			
39	com.gtlive.android.gphone/u0a104	10			
40	com.sun.android.easylabs/u0a176	10			
41	com.sun.android.upc.avenue/u0a119	10			
42	com.unkl.downloadprovidervcumeng/u0a165	10			
43	com.hulukaigametools/u0a122	11			
44	com.autonavi.minimapbizbehaviorservice/u0a121	12			
45	com.sun.unkl/u0a116	13			
46	com.wandoujiaphoenix2/u0a96	16			
47	com.android.vending/u0a31	28			
48	com.study/u0a124	37			
49	com.k123.yqdy/u0a167	54			
50	com.google.android.gms/u0a12	94			
	Total	404			

Fig. 10. Optimization of processing data of pre/post service startup.

The proposed method of reducing unnecessary memory can effectively reduce unnecessary process startup and reduce unnecessary memory overhead.

4.2.3 Response Time

We short the response time of response and improve system performance by counting the time difference between operation and interface display. During the experiment, we recorded video for each of two devices in the following operations:

(1) Leave it on for 10 min after turning it on;
(2) Open the application list and click the app's icon to record the time point when the user pressed the icon as the start time.
(3) After the system displays the application interface, record the time point of the display interface as the end time.
(4) According to the application sequence in the application list, cycle step 2 and 3. We respectively count the click time point and interface display time point of the top 15 applications. Then we calculate the length of time the system responds to the user operation. Results are shown in Fig. 11.

No.	APP	"T-original" Device (Before Optimization)			"T-optimized" Device (After Optimization)		
		Click time	Display time	The display time of Page (Unit / Second)	Click time	Display time	The display time of Page (Unit / Second)
1	Connotations	00:00:04.05	00:00:10.22	6.561	00:00:16.22	00:00:17.24	1.066
2	Watermelon Video	00:00:21.18	00:00:24.21	3.099	00:00:32.20	00:00:32.29	0.297
3	China Mobile	00:01:27.03	00:01:46.19	19.528	00:00:40.26	00:00:41.06	0.033
4	Alipay	00:02:05.04	00:02:07.28	2.792	00:00:55.04	00:00:57.24	2.066
5	YouKu	00:02:21.05	00:02:38.21	17.561	00:01:08.16	00:01:09.27	1.363
6	iFly IME	00:03:20.18	00:04:08.00	47.396	00:01:21.04	00:01:25.18	4.462
7	Xiao Yuan Sou Ti	00:04:25.12	00:04:31.24	6.396	00:01:31.24	00:01:32.08	0.462
8	Xian Yu	00:05:43.15	00:06:10.01	26.528	00:01:48.22	00:01:49.11	0.627
9	XimalayaFM	00:07:15.10	00:09:45.21	150.363	00:03:13.04	00:03:13.17	0.429
10	WeChat	00:11:44.27	00:14:16.14	151.561	00:03:31.13	00:03:34.13	3
11	Weibo	00:17:16.03	ANR 00:20:25.06	189.099	00:03:43.07	00:03:44.02	0.825
12	NetEase Music	00:21:36.08	System Crash	199	00:03:52.29	00:03:53.23	0.792
13	Table Game	00:02:18.09	00:04:48.13	150.132	00:04:17.26	00:04:18.27	1.033
14	Tencent Video	00:05:32.23	00:07:55.13	144.066	00:05:09.21	00:05:10.12	0.693
15	Tencent Comic	00:09:56.26	00:11:48.26	112	00:05:20.01	00:05:20.29	0.924
The average time		——	——	81.7	——	——	1.2

Fig. 11. Time statistics of T-original and T-optimized.

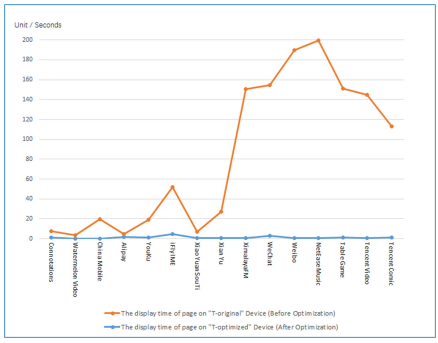


Fig. 12. Time comparison chart of T-original and T-optimized.

For ease of observation, as shown in Fig. 12.

It can be found from the test results that before optimization ("T-original" device), response for a long time. Especially in the case of 7 applications after startup, the display interface takes 2–4 min and even the application does not respond (Weibo) or system crash (Neteasy Music) and other serious problems. The optimized device ("T-optimized" device) has an average response time of 1.2 s. The response time of only 2 applications is longer, around 3–4 s, and no application is unresponsive or system crash.

By increasing the size of the free memory and reducing the application self-starting or association startup, the system can effectively shorten the response time of the user. In addition, it is also found that limiting the application self-starting or association startup significantly reduce the number of broadcasts and services.

Based on the above verification, we demonstrate the efficiency of low memory recovery mechanism and avoid the device being in a low memory state for a long time.

4.3 Discussion

It is worth mentioned that this solution is optimized for the lack of low memory management mechanism, which is different from the third-party application for optimizing mobile phone memory, such as 360 security guard. At present, the three-party application on the market mainly provides two types of functions: optimizing the memory and limiting the application self-starting. The former has three shortcomings:

1. They terminate all application processes, including those are in use. Our method terminates an application that the user does not want to close, and the application reload when the application is recently used.
2. The memory can be cleaned only when the user uses the one-click function to clear the memory. Therefore, real-time monitoring and optimization cannot be achieved, and the effect is not durable. Although some applications provide timing cleanup, which lead to frequent reloads of the switch application. The program optimizes memory from within the system, with higher authority and obvious effects. And the process will be terminated only when the system is in a low memory state, which has less impact on the user.
3. It cannot be customized to the product, and is not the best choice. We adjust the low memory threshold appropriately according to the memory parameters of the product. Our solution is optimized from the inside of system, so it can be customized.

5 Conclusions

Mobile applications usually can only access limited amount of memory, especially in low memory situation. In this paper we analyze the principle of the Low Memory Killer mechanism and point out the reasons for the low recovery efficiency and error recovery process of LMK. After that, we propose the optimization scheme for avoiding the low memory state. Experimental results show that our methods effectively increase the size of free memory, reduce the phenomenon of application self-startup and association startup. In the future, we plan to further extend our approach suitable for more applications. Besides we will consider new combined methods or caching strategies to optimize memory usage.

Acknowledgments. This work is supported by the National Natural Science Foundation of China (No. 61672046).

References

1. Annuzzi, J., Darcey, L., Conder, S.: Introduction to Android Application Development: Android Essentials. Addison-Wesley Professional (2015)
2. Nomura, S., Nakamura, Y., Sakamoto, H., Hamanaka, S., Yamaguchi, S.: Improving choice of processes to terminate in Android OS. GCCE, pp. 624–625 (2014)
3. Gorman, M.: Understanding the Linux Virtual Memory Manager. Prentice Hall Professional Technical Reference (2004)

4. Mauerer, W.: Professional Linux Kernel Architecture. Wiley Publishing, Inc. Technical Reference (2008)
5. Joongjin, K., et al.: Optimization of out of memory killer for embedded Linux environments. In: Proceedings of the 2011 ACM Symposium on Applied Computing. ACM (2011)
6. Purkayastha, D.S., Singhla, N.: Android optimization: a survey. Int. J. Comput. Sci. Mob. Comput.-A Mon. J. Comput. Sci. Inform. Technol. **2**(6), 46–52 (2013)
7. Zhao, H., Chen, M., Qiu, M., Gai, K., Liu, M.: A novel pre-cache schema for high performance Android system. Future Gener. Comp. Syst. **56**, 766–772 (2016)
8. Vásquez, M.L., Vendome, C., Tufano, M., Poshyvanyk, D.: How developers micro-optimize Android apps. J. Syst. Softw. **130**, 1–23 (2017)
9. Lee, J., Lee, K., Jeong, E., Jo, J., Shroff, N.B.: CAS: context-aware background application scheduling in interactive mobile systems. IEEE J. Sel. Areas Commun. **35**(5), 1013–1029 (2017)
10. Nagata, K., Yamaguchi, S., Ogawa, H.: A Power Saving Method with Consideration of Performance in Android Terminals. UIC/ATC, pp. 578–585 (2012)
11. Nomura, S., Nakamura, Y., Sakamoto, H., Hamanaka, S., Yamaguchi, S.: Improving choice of processes to terminate in Android OS. GCCE 2014, pp. 624–625 (2014)
12. Li, C., Bao, J., Wang, H.: Optimizing low memory killers for mobile devices using reinforcement learning. In: 13th International Wireless Communications and Mobile Computing Conference (IWCMC), pp. 2169–2174 (2017)
13. Kim, S.-H., Jeong, J., Kim, J.-S., Maeng, S.: SmartLMK: a memory reclamation scheme for improving user-perceived app launch time. ACM Trans. Embedded Comput. Syst. **15**(3), 47:1–47:25 (2016)
14. Kim, J.H., et al. A novel android memory management policy focused on periodic habits of a user. Ubiquitous Computing Application and Wireless Sensor, pp. 143–149. Springer, Dordrecht (2015)
15. Yang, C.-Z., Chi, B.-S.: Design of an Intelligent Memory Reclamation Service on Android. TAAI 2013, pp. 97–102 (2013)
16. Zhang, X., Tan, Y., Zhang, C., Xue, Y., Li, Y., Zheng, J.: A code protection scheme by process memory relocation for android devices. Multimedia Tools Appl. **77**(9), 11137–11157 (2017). https://doi.org/10.1007/s11042-017-5363-9
17. Yu, Q., et al.: Two-level rejuvenation for android smartphones and its optimization. IEEE Trans. Reliab. (2018). https://doi.org/10.1016/j.ress.2017.05.019
18. Kumar, V., Trivedi, A.: memory management scheme for enhancing performance of applications on Android. In: 2015 IEEE Recent Advances in Intelligent Computational Systems (RAICS). IEEE (2015)
19. Liang, Y., Li, Q., Xue, C.J.: Mismatched Memory Management of Android Smartphones. HotStorage (2019)
20. Amalfitano, D., Riccio, V., Tramontana, P., Fasolino, A.R.: Do memories haunt you? An automated black box testing approach for detecting memory leaks in android apps. IEEE Access **8**, 12217–12231 (2020)
21. Lee, S.S., Lee, D.H., Lee, D.K., Kang, H.H., Ahn, C.A.: A Novel Mobile Robot Localization Method via Finite Memory Filtering Based on Refined Measurement. SMC 2019, pp. 45–50 (2019)
22. Ryusuke, M., Yamaguchi, S., Oguchi, M.: Memory consumption saving by optimization of promotion condition of generational GC in android. In: 2017 IEEE 6th Global Conference on Consumer Electronics (GCCE). IEEE (2017)
23. Yang, Q., Peng, G., Gasti, P., Balagani, K.S., Li, Y., Zhou, G.: MEG: memory and energy efficient garbled circuit evaluation on smartphones. IEEE Trans. Inform. Forensics Secur. **14**(4), 913–922 (2019)

24. Kalkov, I., Franke, D., Schommer, J.F., Kowalewski, S.: A Real-Time Extension to the Android Platform. JTRES 2012, pp. 105–114 (2012)
25. Song, W., Kim, Y., Kim, H., Lim, J., Kim, J.: Personalized optimization for android smartphones. ACM Trans. Embedded Comput. Syst. **13**(2 s), 60:1–60:25 (2014)
26. Yang, C.-Z., Chi, B.-S.: Design of an Intelligent Memory Reclamation Service on Android. TAAI 2013, pp. 97–102 (2013)
27. Ju, M., Kim, H., Kang, M., Kim, S.: Efficient memory reclaiming for mitigating sluggish response in mobile devices. ICCE-Berlin 2015, pp. 232–236 (2015)
28. Ahn, S.: Automation of Memory Leak Detection and Correction on Android JNI. MobiSys 2019, pp. 533–534 (2019)
29. Yang, S.J., Choi, J.H., Kim, K.B., Bhatia, R., Saltaformaggio, B., Xu, D.: Live acquisition of main memory data from Android smartphones and smartwatches. Digital Invest. **23**, 50–62 (2017)
30. Kassan, S., Gaber, J., Lorenz, P.: Autonomous energy management system achieving piezoelectric energy harvesting in wireless sensors. Mob. Netw. Appl. **25**(2), 794–805 (2019). https://doi.org/10.1007/s11036-019-01303-w
31. Maiti, A., Chen, Y., Challen, G.: Jouler: A Policy Framework Enabling Effective and Flexible Smartphone Energy Management. MobiCASE 2015, pp. 161–180 (2015)