Computers and Electrical Engineering 71 (2018) 223–236

Contents lists available at ScienceDirect

Computers and Electrical Engineering

journal homepage: www.elsevier.com/locate/compeleceng

Springbok: An App deployment accelerator for Android smart devices[☆]

Shaoyong Li[*,a], Yaping Liu[b,a], Zhihong Liu[c], Ning Hu[c]

[a] Central South University, Changsha, Hunan, PR China
[b] Guangzhou University, Guangzhou, Guangdong, PR China
[c] National University of Defense Technology, Changsha, Hunan, PR China

ARTICLE INFO

Keywords:
Performance
Android OS
Mobile system
Energy optimization

ABSTRACT

Mobile smart devices with Android OS have been widely used in recent years. Developers worldwide are building various complex Apps that require increasing resources and energy. However, as the installation process becomes longer, users must wait for more time before using a new App or upgrading an old one. To address this issue, we propose Springbok, a system that speeds up App deployment for Android devices. By placing the final optimized target file into the Android Package, Springbok directly skips the optimization task, which is the most time-consuming and energy-consuming operation in the installation process. Finally, we implement Springbok and evaluate it with different Android devices. The experimental results show that, in our experimental environment where the WiFi access rate is 65Mbps, Springbok can save time and energy consumption by up to 81% and 86%, respectively.

1. Introduction

With the capability of running a variety of Apps, Android OS has many advantages over extensibility, simplicity and flexibility. Therefore it attracts a lot of attention from users and manufacturers. However, while the applications are becoming more and more complex, App installation is not only time-consuming but also power-hungry. For example, if users want to install an App, they may need to wait for a few seconds or even minutes. If users want to upgrade their device's operating system, they may need to wait for dozens of minutes or even hours. Moreover, users will need to ensure that their devices have sufficient power (for example, 30% or more) as specified by the manufacturer. If not, they will need to connect their devices to a charger. Therefore, even if some Apps are rarely used, the user is more willing to keep them in the device, thereby affecting the device's performance. If we can speed up the installation process, users are more likely to uninstall unused Apps rather than keep them on the devices. This not only reduces the waiting time for App installation, but also improves the fluency of the system, which significantly improves user experience. Therefore, accelerating App deployment on Android platform is meaningful and practical task.

By comparing Android smart phones and wearable devices via the experiments detailed in Section 5.2, the most resource-intensive and time-consuming part of the entire App installation process is translating the "dex" bytecode of the Android PacKage (APK) file into optimized bytecode or machine code [1]. This step has the most performance cost of the App installation process. It prolongs App installation, slows down the system and affects user experience seriously. Research into rapid installation like Snapcode, from

☆ Reviews processed and recommended for publication to the Editor-in-Chief by Guest Editor Dr. I. H. Shanavas.
* Corresponding author.
E-mail addresses: lishao378@sohu.com, 282803936@qq.com (S. Li), yp.liu@csu.edu.cn (Y. Liu), zhihonglzh2005@163.com (Z. Liu), huning@nudt.edu.cn (N. Hu).

https://doi.org/10.1016/j.compeleceng.2018.07.052
Received 17 May 2018; Received in revised form 1 July 2018; Accepted 31 July 2018

Tsinghua University, also points out that the Android App installation process uses the most energy consumption [2].

Based on these observations, we propose Springbok, an App deployment accelerator for Android smart and wearable devices. By making a small modification to Android's original application installation process, and placing the optimization target file into the App installation package, we can then skip localized compilation step while maintaining synchronization with the official source code; this is done since localized compilation is the most time-consuming step during App installation. Instead of this step, we build a rapid deployment system with the cloud App store running on the server and sharing the optimized target files. As a result, the system can significantly reduce time and energy consumption.

The contribution of this work are as follows:

a) We reduce the time and energy consumption of Android devices while installing Apps. Moreover, the compatibility of the App installation package and the efficiency of the installation process can be simultaneously assured by placing an optimized target file into the APK package.
b) We improve the general service method of the App Store. By carrying the terminal system signatures, the App store client can acquire suitable improved APK packages.
c) We propose a solution for sharing optimized target files between devices. The optimized target files generated by local devices will be uploaded to the cloud App store. Then the cloud App store will generate an improved version of the installation package for all the terminals with the same system signature, thereby achieving a wider range of quick installation processes.

The rest of the paper is organized as follows. Section 2 gives a brief introduction of the background. Section 3 describes the framework and the implementation of Springbok on the mobile platform. Section 4 discusses the analysis and the model of the performance of the deploying applications. Section 5 shows the experiments and evaluation of Springbok and Section 6 discusses related work. Finally, in Section 7, we draw the conclusions and present the ideas for future work.

2. Background

This section briefly introduces the format of Android's installation package defined by Google and discusses some basic concepts of the Apps' life cycle on Android platforms.

2.1. The structure of an APK file

As shown in Fig. 1, the Android App installation package is a file with the suffix "apk", which is a zip archive that consists of several files and folders. The AndroidManifest.xml file stores meta-data such as package name, required permissions, definitions of one or more component like activities, services, broadcast receivers or content providers, minimum and maximum version support, libraries to be linked, etc. The folder "res" stores icons, images, string/numeric/color constants, UI (User Interface) layouts, menus and animations compiled in binary. The folder "assets" contains non-compiled resources. The executable file "classes.dex" stores the Dalvik bytecode that is later executed on a Dalvik virtual machine [1]. The META-INF stores the App developers certificate to verify the third-party developer's identity. The final file is called resources.arsc. It records the mapping between the resource file and the resource ID [3].

2.2. The installation of an Android App

Before running an Android App, there are two steps users need to accomplish: downloading and installation steps.

In general, the downloading step of an App is often achieved using the http protocol to download the installation package from a remote server. The time consumption depends on the network speed, the size of the installation package, and the re-transmission after network instability.

The installation stage is very complicated, which can be seen in Fig. 2. In the application layer, App managers such as Google Market and built-in "PackageInstaller" on the Android system are triggered by a user's click (or other actions) to get the APK file that will be installed. Then, they send installation request to the middle layer ("PackageManager") on the Android platform. PackageManager submits the request to the service layer process ("PackageManagerSerivce") via a method called "bind". The

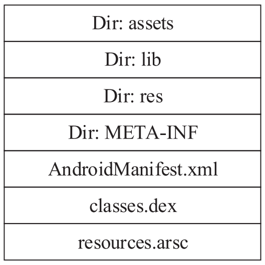


Fig. 1. Structure of APK file.

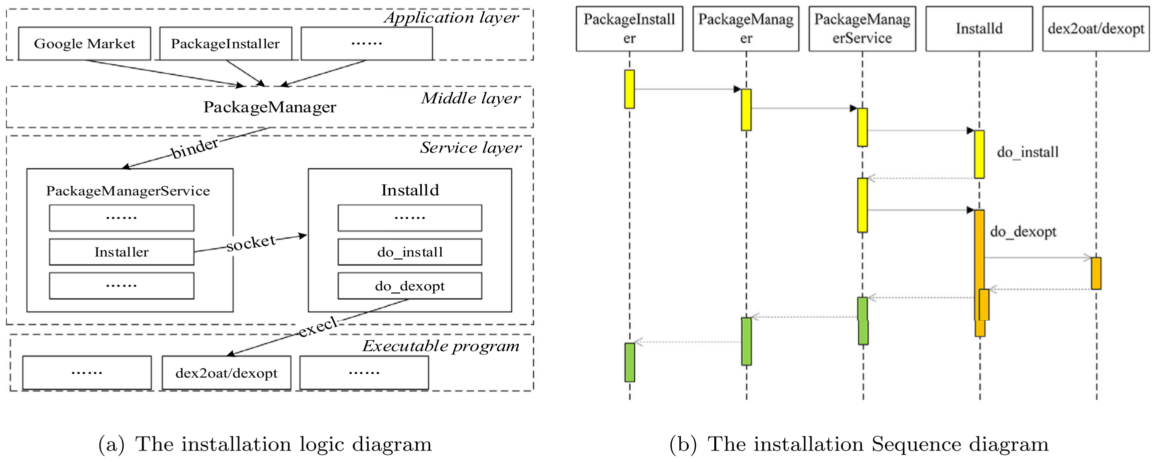


Fig. 2. App installation process of Android.

PackageManagerSerivce uses an internal process-based communication method ("socket") to call the system service "installd". Then, "do_install" function is executed to detect/create a file, to change permissions and other similar operations. Finally, "installd" executes the "do_dexopt" that calls the external program "dex2oat" or "dexopt" via "execl", to perform the file optimization operation and generate the final optimized target file that will be used during the running stages of the Apps.

Thus, the installation stage can be divided into two sub parts.

- **The first part ("PM&do_install").** This part contains the Service Package Manager and the function "do_install". This function only uses some simple operations and a small amount of time. Specifically, it copies the APK file to "/data/app/" directory and unpacks the lib directory into another directory like "/data/data/ < package name > /". It does not process the "assets" and "res" directories or the file "resources.arsc" but they are all still contained in the APK package. When a certain library or function needs to be run, Android obtains the appropriate resources in "assets" and "res" directories via the mapping information in resources.arsc.
- **The second part ("do_dexopt").** This part contains only the function "do_dexopt", performs the optimization task, compile the bytecode file "classes.dex" into the native code, which is stored in the "/data/dalvik-cache/ < arch > " (< arch > represents the CPU architecture code) directory. So, this part is very complicated and time-consuming.

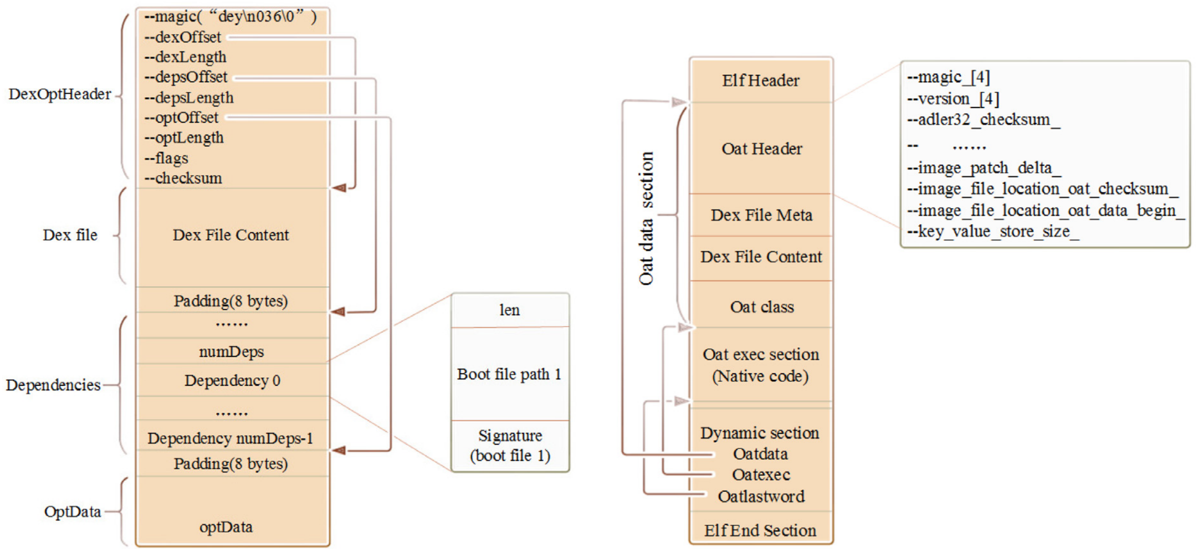
2.3. The optimized target file in Android

As analyzed in Section 2.2, the most important part of the installation of an App is compiling the Dalvik Executable (DEX) bytecode in the Apk file to the final optimized target file used during program execution.

As we all know, before version 4.4, Android used the Dalvik virtual machine to run Apps. This code format is called the Dalvik bytecode. For speeding up the execution, the Dalvik bytecode, which is the DEX file in the APK, will be compiled into optimized DEX (ODEX) file in the installation. Fig. 3(a) shows the structure of Android's ODEX file [4].

The ODEX file can be described as a superset of the DEX file. It is the optimized DEX bytecode file used to run an application on the Dalvik virtual machine of Android systems. The "Dependencies" is a list of dependent libraries for the ODEX file. It is also a list of items stored in the system environment variable BOOTCLASSPATH. The dependency libraries are also programs that run under the Dalvik virtual machine. They will also be optimized and the result will be saved onto ODEX files. These specified libraries are preloaded into the Dalvik virtual machine. As a result, they can be used directly by the Apps. The same libraries are used when the DEX files of Apps are optimized. The main information is saved in "Dependencies" and describes the optimized files of these library files, which contains the size, the absolute path and the signature of the corresponding files. In the terminal system, all the Apps (including libraries specified in BOOTCLASSPATH) are optimized in the same environment using the same version of the Dalvik virtual machine as the same libraries in the BOOTCLASSPATH. Therefore, if the signatures of the optimized files, that are compiled from the libraries specified in BOOTCLASSPATH, are the same, all the ODEX files generated from other Apps are necessarily identical. Therefore, the optimized ODEX file can be used on all the devices with the same terminal system signature.

But the ODEX file is still a Dalvik bytecode file; thus, it still needs to be dynamically compiled into the native code and loaded into memory before each run. As a result, the loading of an App is time-consuming. To improve the loading performance of Apps, starting with Android V4.4, Google uses the Android Runtime (ART) mode instead of the Dalvik virtual machine [1]. In ART mode, one App needs to be compiled from the intermediate "dex" code into the native code in the installation stage, which is called Ahead-Of-Time (AOT) [5]. Note that Google's App installation package is still an APK file that contains "dex" bytecode. When installing an App, the bytecode needs to be directly compiled into machine code [5], so the installation process in the ART mode takes even more time than that in the Dalvik virtual machine [1]. In the loading stage, the local machine code can be executed directly. This step saves the time spent on Just-In-Time (JIT) conversion before the execution of an App. Fig. 3(b) shows the structure of OAT file. For Android systems



(a) The Structure of ODEX file (b) The Structure of OAT file

Fig. 3. Final optimized target file in Android.

running in ART mode, the OAT file is the ultimate executable file used in the Apps' running stage, and in fact, it is a special ELF file. Thus, at the outermost level, the OAT file has the structure of a general ELF file. For example, the OAT file has a standard ELF file header and a section for the file content description. As a private ELF format for Android, the OAT file contains two special segments: "oatdata" and "oatexec". The former contains DEX file contents that are used to generate native machine code. The latter contains the generated native machine code. The relationship between them is described by the OAT header stored in front of the "oatdata" segment.

All we care about here are the four member variables of the OAT header. The "image_file_location_data_" indicates the address of the file path used to create the image. The "image_file_location_size_" represents the size of the file path used to create the image. The "image_file_location_oat_data_begin_" represents the "oatdata" section of the OAT file used to create the image. The "image_file_location_oat_checksum_" represents the checksum of the OAT file used to create the image space. These four member variables document the information of how an image space is created and which files it relies on. For the external applications in Android, the file used to create image space is the system OAT file, which is the initial compilation ("/system/framework/ < arch > / boot.oat") or "/data/dalvik-cache/system@framework@boot.art@classes.dex" that is compiled during the first system boot. The "image_file_location_oat_checksum_" is boot.oat's header checksum "adler32_checksum_". Therefore, the generated OAT machine code of an App can be the same on all devices with the same "adler32_checksum_" value of the "boot.oat" above.

3. Design and implementation

This section introduces our App deployment accelerator for Android smart devices, which is named Springbok. Springbok modifies the Android system service process 'installd" and the Android system APK package format to improve the application's installation process. Besides, we design an cloud App store and App store client to assist the rapid deployment process. As a result, we can ensure the compatibility of the App installation packages, and improve the App installation speed effectively.

3.1. Architecture of Springbok

Springbok's architecture is shown in Fig. 4. There are three components: the cloud App store, the App store client and the improved system service daemon "installd". The cloud App store is located on the server side. It is composed of an application management module, a machine code integration module and an application center database. In parallel, the cloud App store provides three services to all the connected mobile smart devices: the installation package downloading service, the available App listing service, and the optimized target file uploading service. When the application management module receives a downloading request from an App store client, the module will be assigned a parameter named "TSS" (TSS is the abbreviation of Terminal System Signature, which is illustrated in Section 3.2). Then it queries from the application center database, whether there is an improved APK file for the giving TSS. If the improved APK file exists, the application management module gives it to the App store client. Otherwise, the application management module gives the original APK file to the App store client. The improved APK file is a new extended format APK that resembles the Android's original APK file. It contains the optimized target file required by the improved daemon "installd" to support the rapid deployment. We will illustrate details of an improved APK file in Section 3.3.

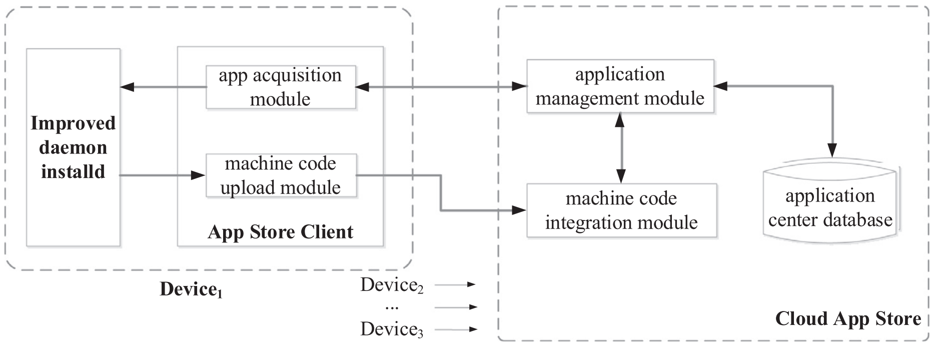

Improved daemon installd
app acquisition module
machine code upload module
App Store Client
Device₁
application management module
machine code integration module
application center database
Device₂
...
Device₃
Cloud App Store

Fig. 4. Architecture of Springbok.

In each Android smart device, there is an App store client that is composed of an App acquisition module and a machine code upload module. The App store client is used to display the available App list on the cloud App store, download improved APK files supporting rapid deployment and upload the optimized target file which is compiled in the native device.

The modified daemon "installd" is running on each Android smart device, which is used to support the new APK format file that contains the optimized target file. By using this daemon, we can achieve quick App deployment. The details of this daemon are shown in Section 3.4.

3.2. Terminal system signature

The terminal system signature (TSS) refers to the unique identification of the type of Android smart device with the same identical hardware and system software. As a result, the optimized target files used in the running stage can be the same (similar) between the Android smart devices with the same TSS. According to the above definition, the terminal system signature can be determined by different methods. Hence, according to the internal characteristics of the Android system analyzed in Section 2.3, we use the signatures of the libraries specified in BOOTCLASSPATH, which is in the Dalvik virtual machine and the "adler32_checksum_" value of the "boot.oat" in the ART mode as the TSS of an Android device. And the final machine code can be same on all the devices with the same TSS.

3.3. The improved APK file

In the definition of an Android original APK package format, for being compatible with a wide range of hardware devices, Google only builds the Dex byte code file into the APK file. When installing an App in devices, the Dex file should be compiled again to generate the final optimized target machine code file, which is OAT format file in the ART virtual machine or ODEX format file in the Dalvik virtual machine. Thus, for the customized series of equipment based on Android platform, we can use the final optimized target file directly for the limited types of hardware. Then, during the installation process, we could use this file to skip the optimization process in the device to accelerate the App installation.

Fig. 5 shows the structure of the improved APK file. It is also a compressed file with the suffix name "apk", which resembles the original APK file described in the background section. Compared with the original APK file, the optimized APK file has only adds optimized target file "target.file". When a user needs to use a new App, the App store client will bring the TSS to the cloud App store to download the App installation package. The latter uses the App name and TSS to query whether there is a corresponding improved APK file in the cloud. If so, the cloud App store provides the improved APK file. If not, it will provide the original APK file and request that client upload the compiled machine code file with the user's permission. Thus, even if ordinary devices do not support rapid installation, they can get these improved APK files to work normally. Because the improved APK file contains all the content of the original APK file, when installing these Apk files, except for downloading, the original Android installation and compilation process

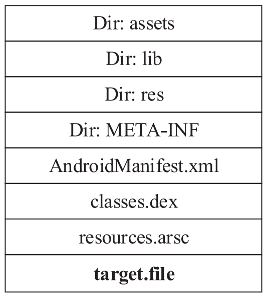


Fig. 5. Structure of the improved APK.

will be performed without increasing the time and energy consumption. Thus, the APK files can be installed normally on an ordinary non-custom platform, which improves its compatibility.

3.4. *The modified daemon "installd"*

Compared to the original "installd" shown in Fig. 2, the improved version only modifies the implementation of "dex_opt" function, as shown in Algorithms 1 and 2. The improved components are as follows:

a. **Changing the optimization process that generates the optimized target file.** After the optimized target file is created, the file's permissions are changed and the read/write handler of the file is obtained. The improved "installd" does not directly perform the native compilation process, but it executes Line 2 in Algorithm 2 to determine whether the desired APK file is available for the optimized target file. If it is, then the process executes the Line 3 in Algorithm 2 to ensure rapid installation using the target file. This step ends the installation process. If it is not, the "installd" daemon calls the Android built-in executable program to perform the APK file optimization operation, which can generate the target file that should be used to run the App.
b. **Changing the ending of the original installation process.** After the improved "installd" calls the built-in executable program to perform the APK file optimization operation and generate the final optimized target file, it first executes Line 6 in Algorithm 2 to send the "upload target file" request to the machine code uploading module. This is done so that it can upload the target file to the cloud App store. After that, the improved "installd" ends the installation process.

Since the improved "installd" determines whether the installation package contains available optimized target file or not, the local compiler can also be used to perform the installation operation of the normal APKs. Thus, we can ensure its compatibility.

4. Analysis and model

Before using a new App, the user must first deploy it. And as we can see from Section 2.2, the deployment contains two main stages: downloading stage and installation stage. The main operations of the installation stage are composed of installation and optimization, which is done via "PM&do_install" and "do_dexopt," respectively. Thus, the total time consumption of the deployment can be described as:

$$T_d = Size/B + \varphi(Size)/Speed + N/Speed \tag{1}$$

Whereas the total energy consumption can be described as:

$$E_d = p_t{*}Size/B + p_u{*}\varphi(Size)/Speed + p_c{*}N/Speed \tag{2}$$

where

- *Size* indicates the size of the installation package, *B* represents the effective network bandwidth, and p_t indicates the power consumption during downloading.
- φ(*Size*) indicates the number of machine instructions that need to be run during the "PM&do_install" phase, *Speed* indicates the speed of the device, which is denoted by the number of instructions per second. And p_u indicates the power consumption at this phase.
- *N* indicates the number of machine instructions during the "do_dexopt" phase and p_c indicates the power consumption at this phase.

Since Springbok skips the compilation part by using the built-in optimized target file, the new improved APK file is larger than the original APK file. If the increase of the file size is *ΔSize*, the total time in the deployment is reduced to:

$$\Delta T_d = N/Speed - \Delta Size/B - \varphi(\Delta Size)/Speed \tag{3}$$

Thus, total energy consumption is reduced to:

$$\Delta E_d = p_c{*}N/Speed - p_t{*}\Delta Size/B - p_u{*}\varphi(\Delta Size)/Speed \tag{4}$$

Using the new deployment method, if $\Delta T_d > 0$, we can claim that the new method is more time-efficient than the original method. If $\Delta E_d > 0$, we can claim that the new method is more energy-efficient.

5. Evaluation

5.1. *Experimental environment*

We implement a prototype system based on the Android 5.1.1 and conduct experiments on a Galaxy Nexus phone [6] manufactured by Samsung Corporation and on a wearable developing board Newton2_plus [7] from Ingenic Semiconductor Co.Ltd. Next, we connect the testing devices to a Wireless Router via WiFi, connect a laptop via a cable and the Apps are downloaded from the

```
  **Input**: The installation instruction, $C_i$; the APK file to be installed, $A_i$.
  **Data**: $P_i$ is a parameter related to APK file; $P_s$ is a parameter related to current Android system.
  **Output**: The optimized target file that is used in running stage, $TF_i$.
1 $PackageInstaller$ receives $C_i$ and $A_i$;
2 $P_i, P_s \leftarrow PackageManager(C_i, A_i)$;
3 PackageManagerService($C_i$, $A_i$, $P_i$, $P_s$);
4 installd($C_i$, $A_i$, $P_i$, $P_s$);
5 do_install($A_i$, $P_i$, $P_s$);
6 $TF_i \leftarrow DO\_Springbok\_DEXOPT(A_i, P_i, P_s)$;
7 **return** $TF_i$;
```

Algorithm 1. Installation logic of Springbok.

```
  Input: APK file to be installed, $A_i$; parameters related to APK file, $P_i$; parameters related to current Android system, $P_s$.
  Data: $TF_i$ is the optimized target file to be used in the running stage.
1 Initialization: calculates the path of the optimized target file, then creates the file, changes permissions of the file, and gets its
  read and write operation handler;
2 if $IsOptimizedApkFile(A_i)$ then
3     DoRapidOptimization($A_i, P_i, P_s$);
4 else
5     $TF_i \leftarrow$ OldOptimizationProcess($A_i, P_i, P_s$);
6     UploadTargetFile($TF_i$);
7 end
```

Algorithm 2. Function DO_Springbok_DEXOPT.

Table 1
Prepared Apps for Nexus phone. All the Apps are downloaded from Google Play Store [10], and they are all in the "Editors' Choice Apps" in May 09, 2018.

Name	Youtube	Google Search	Google Map	Instagram	Snapchat	Google Play Store
Version	11.01.70	8.1.12	9.76.1	43.0.0	10.25.0	9.2.11
Size(MB)	14.87	60.71	28.69	29.49	63.80	14.21

laptop and installed in the devices.

We chose 6 Apps to test the mobile phone and the development board respectively. The results are shown in the Tables 1 and 2. We use the Android Monitor of Android studio to get the time-consumption and use the Power Monitor [8] to get the devices' power consumption rate [9]. Each result is obtained via the mean value of 20 measurements for each scenario. There is a pause for 30 seconds between two consecutive executions.

5.2. *Performance of the original Android system*

As described in Section 2.2, we profile the time consumption during the downloading and installation stages. Similarly, the two parts of the installation stage "PM&do_install" and "do_dexopt" are also profiled. All they are shown in Tables 3 and 4, where we also calculate the confidence interval of the installing time consumption when the confidence level is 95%.

As we can see from the results, "do_dexopt" is the most time-consuming component. Using the Power Monitor, we record the power consumption of the Nexus phone during the downloading stage, "PM&do_install" part and "do_dexopt" part, which is 2458 mW, 1886 mW and 2487 mW, respectively. So we can see the performance profiles for Nexus phone in Fig. 6, which includes time consumption and power consumption occupation.

Also, we get that the power consumption of the developing board during the downloading stage. The "PM&do_install" part and "do_dexopt" part is 478 mW, 326 mW and 485 mW, respectively. So we can see the performance profiles for developing board in Fig. 7, which includes time consumption and power consumption rates.

From these results we can see that, before users could run an App, regardless of the performance of the devices, the installation stage is rather time-consuming and energy-hungry, which is mainly caused by the process that compiles "dex" bytecode into the final machine code. Because the time consumption of the downloading stage mainly depends on the network environment and the size of the installation package, the installation stage is a bottleneck affecting the user experience, particularly in the wearable device, which has a low computational capacity and a small RAM. Therefore, in the same network environment and for the same device, it is worthy to do research on the optimization in the installation stage to achieve improvement in terms of the time and energy consumption.

5.3. *Performance of Springbok*

As the optimized target file is added into the APK package, the size of the improved APK increases, as shown in Tables 5 and 6.

5.3.1. *Time consumption performance*

We profile the time consumption of the downloading stage and installation stage of all the testing Apps, as shown in Tables 7 and 8.

Then we compare the total time consumption with the original system measured in Section 5.2, as shown in Fig. 8. From these results, we can see that the overall time consumption has been significantly reduced by skipping the optimization phase. The reductions of the total time are from 61.7 to 75.9% and from 69.4 to 81.3% for Nexus phone and developing board, respectively.

5.3.2. *Energy consumption performance*

The experimental results show that, the "do_dexopt" phase takes more CPU resource and consumes more time. By skipping this part, the energy consumption is significantly reduced, which we can see in Fig. 9. The explicit reductions of the total time are from 66.1 to 79.3% and from 76.4 to 86.0% for a Nexus phone and a developing board, respectively.

5.3.3. *Bandwidth and performance*

The time consumption of the downloading stage only relates to the effective network bandwidth. As the current network access speed is getting faster and faster, then we can usually evaluate the overall performance of the application deployment with the performance of the installation stage. Therefore, if the network speed is not a constraint, then we compare the time consumption and

Table 2
Prepared Apps for developing board. All the Apps are the recommended for SmartWatch [11], and they are all the standalone Apps.

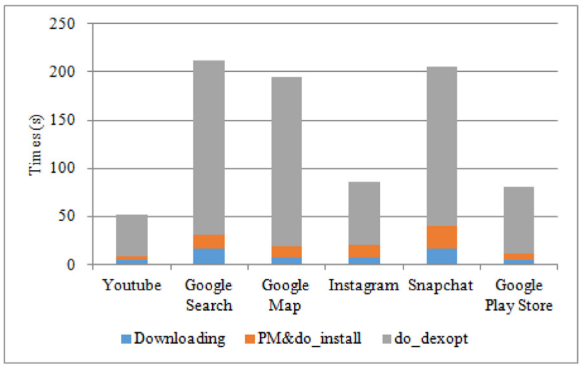
Name	Android Pay	IF by IFTTT	Strava	Google Fit	Foursquare	Wear Mini Launcher
Version	1.33.169	3.6.0	44.0.0	1.80.02	11.6.1	5.8.3
Size(MB)	8.55	6.33	36.24	12.84	19.37	10.05

Table 3
Time consumption of two stages on Nexus Phone using original system.

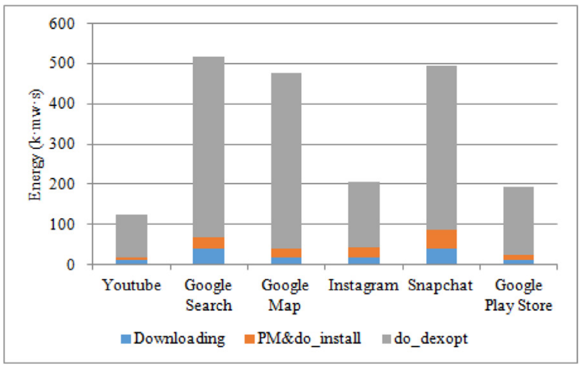
Name	Downloading(s)	Installing		
		Total(s)	PM&do_install(s)	do_dexopt(s)
Youtube	4.14 ± 0.48	47.47 ± 4.76	3.89 ± 0.28	43.58 ± 4.80
Google Search	16.09 ± 0.84	195.90 ± 18.40	14.72 ± 0.77	181.18 ± 17.69
Google Map	7.40 ± 1.00	186.64 ± 17.23	11.25 ± 1.52	175.39 ± 15.88
Instagram	7.69 ± 0.67	78.78 ± 5.37	13.36 ± 1.42	65.42 ± 4.35
Snapchat	16.10 ± 2.32	189.38 ± 12.19	24.42 ± 2.00	164.96 ± 10.43
Google Play Store	4.25 ± 0.79	75.87 ± 3.97	7.07 ± 0.89	68.80 ± 3.53

Table 4
Time consumption of two stages on developing board using original system.

Name	Downloading(s)	Installing		
		Total(s)	PM&do_install(s)	do_dexopt(s)
Android Pay	3.17 ± 0.85	130.92 ± 2.01	9.32 ± 0.28	121.61 ± 1.88
IF by IFTTT	2.25 ± 0.33	121.48 ± 9.46	9.88 ± 1.14	111.60 ± 8.37
Strava	9.73 ± 0.65	259.20 ± 14.43	44.18 ± 2.65	215.02 ± 11.92
Google Fit	3.96 ± 0.24	129.08 ± 5.23	13.00 ± 0.55	116.08 ± 4.80
Foursquare	5.24 ± 0.62	215.06 ± 5.23	21.06 ± 0.45	194.00 ± 4.82
Wear Mini Launcher	3.30 ± 0.40	159.59 ± 5.49	8.62 ± 0.39	150.97 ± 5.18

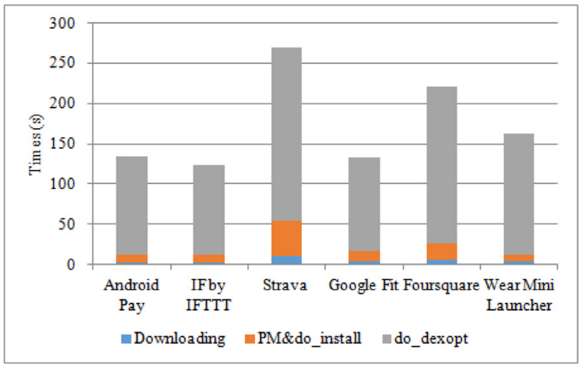


(a) Time consumption of different parts

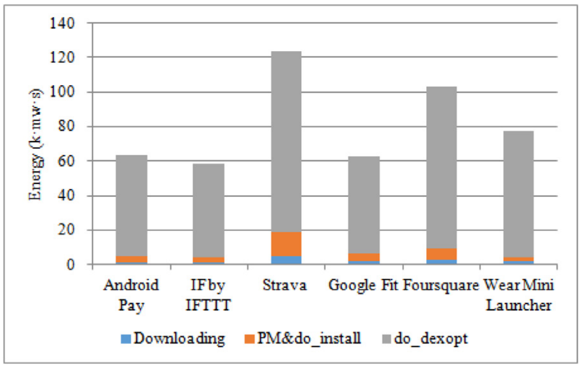


(b) Energy consumption of different parts

Fig. 6. Performance profile of Nexus Phone using original system.



(a) Time consumption of different parts



(b) Energy consumption of different parts

Fig. 7. Performance profile of developing board using original system.

Table 5
Sizes comparison of the original APK and the improved APK for Nexus Phone.

	Youtube	Google Search	Google Map	Instagram	Snapchat	Google Play Store
Old Size(MB)	14.87	60.71	28.69	29.49	63.80	14.21
New Size(MB)	26.38	103.87	69.14	45.19	10.24	29.41
Increased(%)	77.3	71.1	141.0	53.2	57.1	107.0

Table 6
sizes comparison of the original apk and the improved apk for developing board.

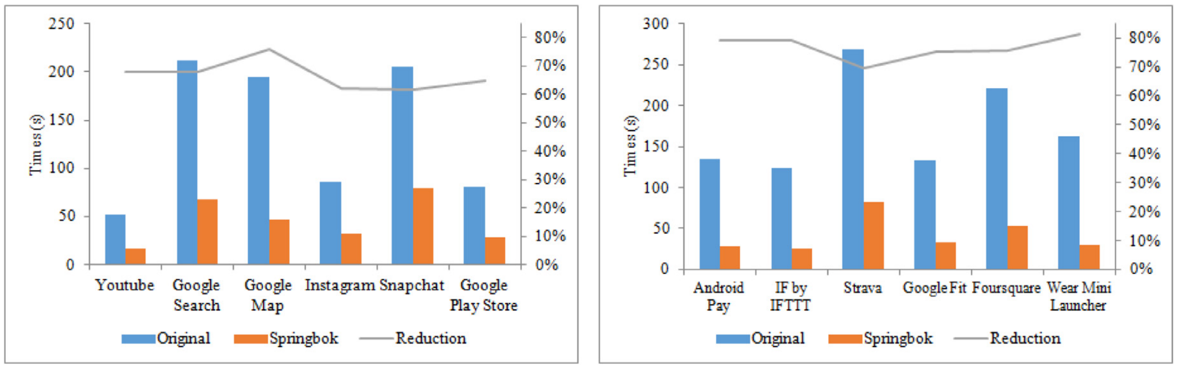
	Android Pay	IF by IFTTT	Strava	Google Fit	Foursquare	Wear Mini Launcher
Old Size(MB)	8.55	6.33	36.24	12.84	19.37	10.05
New Size(MB)	18.63	14.42	52.87	21.82	33.94	21.75
Increased(%)	117.9	127.7	45.9	70.0	75.2	116.4

Table 7
Time consumption on Nexus Phone using Springbok.

Name	Downloading(s)	Installing		
		Total(s)	PM&do_install(s)	do_dexopt(s)
Youtube	7.07 ± 0.67	9.61 ± 0.77	9.03 ± 0.82	0.58 ± 0.13
Google Search	27.45 ± 4.04	40.62 ± 3.17	38.20 ± 3.21	2.42 ± 0.26
Google Map	17.50 ± 3.21	29.34 ± 1.67	27.76 ± 1.45	1.58 ± 0.39
Instagram	11.27 ± 1.25	21.33 ± 2.69	19.77 ± 2.26	1.57 ± 0.51
Snapchat	33.74 ± 6.63	45.04 ± 5.91	43.51 ± 5.41	1.53 ± 0.52
Google Play Store	13.47 ± 4.27	14.65 ± 2.17	13.57 ± 2.11	1.08 ± 0.12

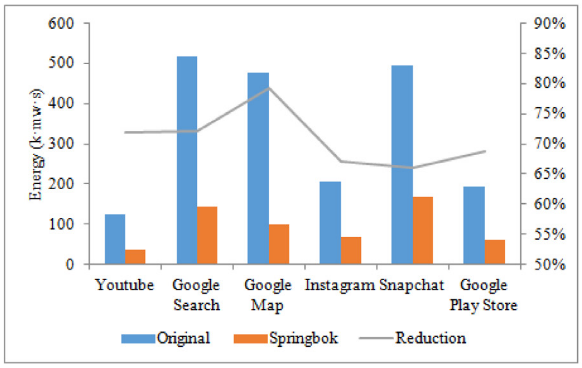
Table 8
Time consumption on developing board using Springbok.

Name	Downloading(s)	Installing		
		Total(s)	PM&do_install(s)	do_dexopt(s)
Android Pay	5.56 ± 0.50	22.36 ± 0.77	20.82 ± 0.50	1.42 ± 0.15
IF by IFTTT	4.58 ± 0.16	21.24 ± 1.20	19.54 ± 0.90	1.38 ± 0.47
Strava	14.00 ± 1.16	68.27 ± 2.84	67.04 ± 2.54	1.59 ± 0.54
Google Fit	6.20 ± 0.45	26.47 ± 0.63	24.56 ± 0.51	1.96 ± 0.29
Foursquare	10.49 ± 1.19	43.32 ± 0.89	41.61 ± 0.90	1.91 ± 0.23
Wear Mini Launcher	5.92 ± 0.33	24.56 ± 0.93	22.89 ± 0.74	1.84 ± 0.17

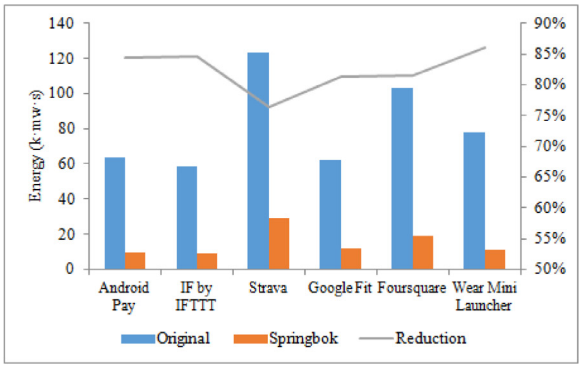


(a) Time Consumption on Nexus phone (b) Time Consumption on the developing board

Fig. 8. Time consumption comparison of original system and Springbok.



(a) Energy Consumption on Nexus phone



(b) Energy Consumption on the developing board

Fig. 9. Energy consumption comparison of original system and Springbok.

energy consumption of the deployment process of Springbok and the original system. Then, by considering the limited network speed, we analyze the effect of network speed on the effect of Springbok optimization.

In the actual WiFi environment with access speed of 64Mbps, we carry out experiments in four situations: no speed limit, a speed limit 2MB/s, a speed limit 1MB/s and a speed limit 512KB/s. With different speed limits, there are different power consumption rates for both devices. On the Nexus phone, the power consumption is 2458 mW, 2300 mW, 2157 mW and 2093 mW, respectively. On the developing board, the power consumption is 478 mW, 407 mW, 379 mW and 343 mW, respectively. Here we provide the experimental results of two devices using their first prepared App, respectively. The results include a comparison of the time-consumption and energy-consumption, as shown in the Fig. 10.

The results show that, as the speed of the network increases, the performance of the system will increase continuously. According to Eqs. (3) and (4), we can see that Springbok can save time and energy only when the effective network speed reaches a certain performance. This limit will occur when the downloading time is small enough to be negligible.

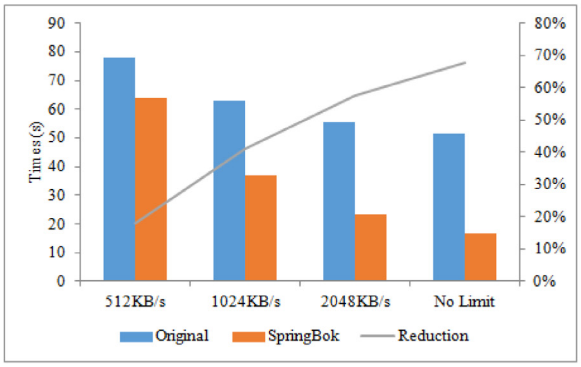
6. Related work

Smartphones and tablet computers have progressively become essential parts of our life. However, these devices are limited in computational resources compared to more advanced processing devices such as personal computers and laptops. To mitigate this problem, cloud computing [12] has become a promising candidate to help resource-constrained devices by offloading heavy applications onto the Cloud. Considerable cloud-based research work have proposed solutions to address their issues of computational power and battery lifetime. Prominent among those are the MAUI [13], the CloneCloud [14] and ThinkAir [15] projects. They optimize the performance of mobile devices by offloading computing tasks to the cloud, such as functions, classes, objects, methods, or threads that are heavily involved in computing. However, they still need the time-consuming and energy-hungry installation process, and can't improve the user experience in the installation stage.

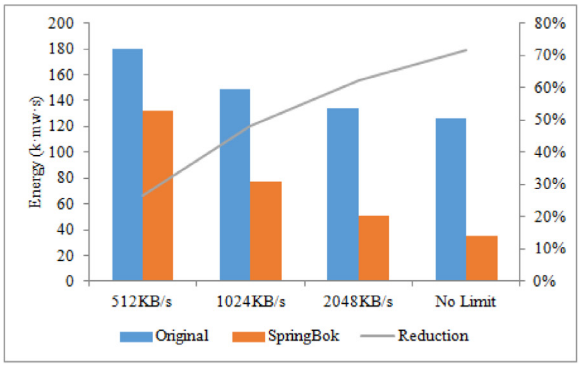
Transparent computing [16] is also a hot topic for light-weight terminals and smart applications such as mobility, portability, user dependency, immediacy, and privacy. It wants to let users enjoy services via on-demand network access with any type of device. Thus, this can be done without needing to know the location of the OSs, middleware or Apps [17]. One of the main ideas of transparent computing is "streaming execution". This is a process where applications can be "fetch when use, run away when over", and can be implemented by some technologies such as virtualization on a traditional PC. On Android system, because the installation process is time-consuming, there is no implementation scheme yet. The rapid installation scheme proposed in this paper can improve the user's ability to choose "fetch when use, run away when over" to a certain extent; this is a simple form of supporting transparent computing in smart mobile devices.

Many industry research studies in App dynamic loading frameworks use the so-called plug-in technology [18]. The purpose of these researches is to reduce the size of the main package, deal with the maximum number of 65,536 methods in each "dex" package [19], increase the function without releasing a new version, repair bugs rapidly, etc. Facebook was the first to develop plug-in technology to solve the failed installation process caused by the growing Facebook App installation package. Facebook uses a very clever way, which is to split the installation package into multiple "dex" files and run App by dynamic loading using DexClassLoader [20]. Later, to solve the similar problem, Google introduces a so-called MultiDex method in Android and standardizes it (The MultiDex is built in Android v5.0 or later) [21]. But those plug-in technologies are only used to solve the problem caused by the Android application executable code's size. They cannot solve the time consumption issue caused by compiling dex code into machine code in the application installation stage.

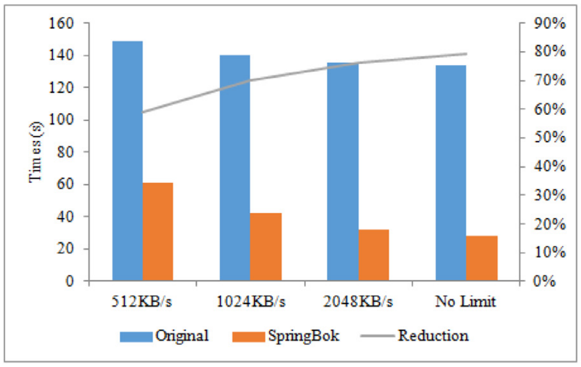
Recently, some new App dynamic loading frameworks were proposed, such as DynamicLoadApk [22], PluginManager [23], DroidPlugin [24]. Then, each APK is divided into host and plug-in parts. And the host part runs normally but the plug-in parts are loaded only when they are needed. Among them, the most representative is DroidPlugin, which is a new Plugin Framework developed and maintained by the Android app-store team at Qihoo 360. This plugin bypasses Android system user authorization and



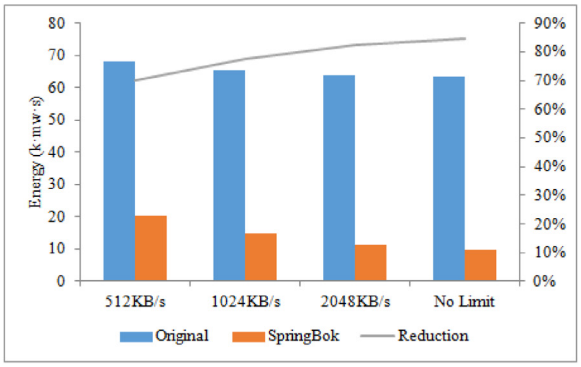
(a) Time consumption on Nexus phone in different bandwidth



(b) Energy consumption on Nexus phone in different bandwidth



(c) Time consumption on developing board in different bandwidth



(d) Energy consumption on developing board in different bandwidth

Fig. 10. Performance with different bandwidth.

enables the host App to run any third-party APK without installation, modification or repackaging. Nevertheless, the process still has drawbacks: First, although users do not need to install the App before running, the dex file should be compiled dynamically each time when running an App for using "DexClassLoader" of the Android Framework, and therefore the App's running speed is affected greatly. Second, the plugin uses reflection technology to provide services to plug-in parts as a middle layer of Android platform, and it will reduce operational efficiency and loss some system features. Besides, its compatibility and API continuity are constrained.

In 2016, Google launched a new capability called "Instant Run" [25] on its Android Studio v2.0. After the first deployment of an application either on a real device or an emulator, the "Instant Run" pushes most codes or resource changes without building a new APK. Android Studio can make this process come true by using one of the three types of code swapping: hot Swap, warm Swap, and cold Swap. The hot Swap is the fastest approach without re-initializing objects in the running application. It is used when the content of an existing method changes. The warm swap requires a restart of the current activity and an image flicker may be noticed. This system is used when resource changes or is removed. The cold swap requires an application restart. The restart function is used to process a long list of edits including adding/removing/changing of annotations, instance fields, static fields, and signature of static methods, parent class, the list of implemented interfaces, and others. The "Instant Run" is not available when deploying simultaneously to multiple devices because it uses different techniques for swapping on various API levels. It is only useful for installed applications in certain cases. It has no optimization effect in a new installation process.

All the technologies mentioned above are not involved in the acceleration of an App's installation. The most related research to our work is Snapcode [2] proposed by Tsinghua University. Snapcode implements a pseudo-installation by constructing a streamlined and small APK that contains only the basic information of ICON and package names. When users really want to use an App, Snapcode will obtain the original APK url and the optimized target file url from the small APK. Then downloading an original APK and loading the pre-compiled and optimized target file from the network, simultaneously, to speed up the App's installation process. The difference between our Springbok and Snapcode is that Springbok provides a systematic framework to accelerate the installation process, while considering how to reduce the increasing downloading cost at the same time. Springbok also makes a good balance between the comparability and its performance optimization.

7. Conclusion and future work

To reduce the waiting time of users who want to use new Apps and to reduce the energy consumption spent by installing Apps, Springbok is designed to speed up App deployment for Android devices. By improving the APK format and the installation process, Springbok can skip the most time-consuming operations and improve the application installation efficiency greatly. Moreover, Springbok can also support installing normal APK files and share the final optimized target files through a cloud App store. We demonstrated improvements brought about by Springbok in a case study that helped save time and energy consumption without error in Android-based prototype devices. The results showed that, in our experimental environment with a WiFi access rate of 65Mbps, Springbok can save up to 81% of the time consumption and 86% of the energy consumption in App's installation stages.

As possible future work, we can make use of the compatibility of the final optimized target files with different terminal system signatures to reduce the server's storage pressure. It is also possible to propose a better target file sharing solution between different devices. In addition, because the final optimized target file of Android system contains the original bytecode in the installation package, we may study how to remove the redundant information to reduce the size of the improved installation package to speed up the downloading stage. Furthermore, since the network speed has an impact on the improvement brought by our system, we can improve the installation optimization process by detecting the device's transmission rate dynamically. When the benefit from the rapid installation stage is greater than the increased consumption of the downloading stage, we provide the optimized installation package to the device; otherwise, we provide the original installation package.

Acknowledgment

This work is supported by National Natural Science Foundation of China (61472438).

References

[1] Yadav R, Bhadoria RS. Performance analysis for android runtime environment. Fifth international conference on communication systems and network technologies. 2015. p. 1076–9.
[2] ChaoWu. Snapcode. Technical report of Tsinghua University. 2017.
[3] Surhone LM, Tennoe MT, Henssonow SF, Phone AD. APK (File format). Betascript Publishing; 2010.
[4] Levin J. Dalvik and art. http://newandroidbook.com/files/Andevcon-DEX.pdf; 2016. Accessed Jun. 15, 2017.
[5] Lim YK, Parambil S, Kim CG, Lee SH. A selective ahead-of-time compiler on android device. International conference on information science and applications. 2012. p. 1–6.
[6] Samsung. Galaxy nexus smartphone. https://en.wikipedia.org/wiki/Galaxy_Nexus; 2012. Accessed May 16, 2018.
[7] I. Semiconductor. Newton2_plus. http://www.ingenic.com.cn/en/?newton/id/13.html; 2015. Accessed May 08, 2018.
[8] M. Solutions. Power monitor. https://www.msoon.com/; 2011. Accessed May 15, 2018.
[9] Google. Performance profiling tools. https://developer.android.com/studio/profile/index.html; 2015. Accessed Jun. 15, 2017.
[10] Google. Google play store. https://play.google.com/store/apps; 2018. Accessed May 09, 2018.
[11] Rao V. 20 must have android wear apps for your smartwatch in 2017. https://www.technorms.com/63810/best-android-wear-apps; 2017. Accessed May 08, 2018.
[12] Kai H, Dongarra J, Fox GC. Distributed and cloud computing: from parallel processing to the internet of things. Morgan Kaufmann Publishers Inc.; 2011.
[13] Cuervo E, Balasubramanian A, Cho D-k, Wolman A, Saroiu S, Chandra R, et al. Maui: making smartphones last longer with code offload. Proceedings of the 8th international conference on mobile systems, applications, and services, MobiSys '10. New York, NY, USA: ACM; 2010. p. 49–62.
[14] Chun BG, Ihm S, Maniatis P, Naik M, Patti A. Clonecloud: elastic execution between mobile device and cloud. Conference on computer systems. 2011. p. 301–14.
[15] Kosta S, Aucinas A, Hui P, Mortier R. Thinkair: dynamic resource allocation and parallel execution in the cloud for mobile code offloading. INFOCOM, 2012 proceedings IEEE. 2012. p. 945–53.
[16] Zhang Y, Guo K, Ren J, Zhou Y, Wang J, Chen J. Transparent computing: a promising network computing paradigm. Comput Sci Eng 2016;19(1):7–20.
[17] Zhang Y, Zhou Y. Transparent computing: a new paradigm for pervasive computing. International conference on ubiquitous intelligence and computing. 2006. p. 1–11.
[18] Singhai A, Ramanujam RS, Bose J, Kumari V. Implementation and analysis of pluggable android applications. 2015 IEEE international conference on signal processing, informatics, communication and energy systems (SPICES). 2015. p. 1–5.
[19] Google. Dexopt fails with "linearalloc exceeded" for deep interface hierarchies. http://code.google.com/p/android/issues/detail?id=22586; 2011. Accessed Jun. 15, 2017.
[20] Schneider D, Hardy Q. Under the hood at google and facebook. IEEE Spectr 2011;48(6):63–7.
[21] Google. Google multidex. https://developer.android.com/tools/building/multidex.html; 2011. Accessed Jun. 15, 2017.
[22] Ren Y.. Dynamicloadapk. https://github.com/singwhatiwanna/dynamic-load-apk/; 2015. Accessed Jun. 15, 2017.
[23] HouKx. Pluginmanager. https://github.com/houkx/android-pluginmgr; 2016. Accessed Jun. 15, 2018.
[24] 360 Droidplugin. https://github.com/Qihoo360/DroidPlugin; 2015. Accessed Jun. 15, 2017.
[25] Google. A guide to using instant run in android studio 2. http://www.techotopia.com/index.php/A_Guide_to_using_Instant_Run_in_Android_Studio_2; 2016. Accessed Jun. 15, 2017.

Shaoyong Li received the B.E. Degree in 2006 and the M.E. Degree in 2008. He is currently pursuing the Ph.D. degree in the Information Science and Engineering Department of Central South University, Changsha, Hunan, China. His main research interests are mobile computing, IOT operating systems, and big-data analysis and applications.

Yaping Liu received the B.S. (1994), M.S. (1997), and Ph.D. (2006) degrees, all in college of computer from the National University of Defense Technology, China. She received her associate professor promotion and full professor promotion in 2004 and 2013, respectively. Her research interests include computer networks and information security.

Zhihong Liu is currently a lecturer in College of Mechatronic Engineering and Automation with National University of Defense Technology. His research interests include big-data analysis, pattern recognition and intelligent system.

Ning Hu is a Professor of Computer Science department at National University of Defense Technology. He received the two Second Class Prize of Chinese State Scientific and Technological Progress Award. He is currently interested in data networking, distributed systems, enterprise and data center networks, network virtualization, and software-defined networking.