A Fast Boot, Fast Shutdown Technique for Android OS Devices

Xia Yang, Peng Shi, Haiyong Sun, and Wenxuan Zheng,
University of Electronic Science and Technology of China

Jim Alves-Foss, University of Idaho

Increasingly complex Android OS applications demand additional software initialization and configuration during startup, which slows system boot time and inconveniences users. The authors propose an approach based on existing snapshot-imaging techniques that can reduce startup time by 80.5 percent and shutdown time by half while avoiding the system-synchronization problem that plagues suspend–resume methods.

The extensibility and availability of Android OS have popularized it as a way to increase the functionality of embedded software in cell phones, digital TVs, and other devices. However, Android has had to provide many features to support converging applications, which has enlarged the code to the point that Android devices currently require an average 30-s bootup time.[1,2] This works against the demand for fast startup, a critical feature for most Android device users who have come to rely on quickly activating their TVs and cell phones.

To address the demand for rapid startup, we investigated the boot and shutdown times for Android OS, which extends our earlier work on optimizing the suspend–resume (SR) technique to reduce Android OS boot time to less than 5 s.[3] Unfortunately, although our earlier technique greatly reduced startup time, it actually increased shutdown time because the suspend operation was implemented for every shutdown or power-off event, and the SD card's write speed is slow. Startup and shutdown times are even longer with these methods when the image is 400 Mbytes or larger. The sidebar "Problems with the Suspend–Resume Technique" describes the limitations of the SR method.

Our current work has resulted in the fast boot, fast shutdown (FBFS) technique, which optimizes the SR method by using improved snapshot imaging that shrinks the snapshot by more than half and requires it to be created only during the system shutdown. The snapshot image includes copies of CPU registers, memory,

0018-9162/16/$33.00 © 2016 IEEE

PROBLEMS WITH THE SUSPEND–RESUME TECHNIQUE

Many efforts to improve an Android device's boot time are based on the suspend–resume (SR) approach, which is essentially a power-management method to restore an image from information stored in auxiliary memory after the system terminates.[1] Information includes memory, task state, and device drivers. Methods typically use SR to optimize boot time,[2,3] or an SR variant that reuses a single saved system snapshot,[4,5] but both of these approaches often have restrictions.

One problem is that the method does not scale to higher Android versions, which are more complex and thus have a much larger snapshot image. Snapshot images of 400 Mbytes or more greatly increase shutdown time because the SR method must write a system image to disk with every power-off, and the write speed of embedded auxiliary memory is very slow. Startup time also increases because the system state must be restored from the image data saved on an external disk. Because data is saved externally, the system could turn off before the save is complete, resulting in the failure to save the generated snapshot image in auxiliary storage.

Another, arguably more serious, obstacle is the synchronization problem. The system state restored from the image is often not the most current state, so any optimized SR method must ensure that the restored state is synchronized with the state in external memory. One method that reuses a system snapshot improves shutdown time[4] but does not solve this problem, so Android device users cannot install, create, delete, or modify any files and applications—an unrealistic limitation for most users.

The fast boot, fast shutdown (FBFS) approach resolves both the snapshot imaging and synchronization problems and still significantly decreases startup and shutdown times relative to other optimized SR methods.

References

1. M. Patrick, "The State of Linux Power Management," *Proc. Linux Symp.*, 2006, pp. 151–163.
2. J.H. Kim et al., "Optimizing the Startup Time of Embedded Systems: A Case Study of Digital TV," *IEEE Trans. Consumer Electronics*, vol. 55, no. 4, 2009, pp. 2242–2247.
3. H. Kaminaga, "Improving Linux Startup Time Using Software Resume," *Proc. Linux Symp.*, 2006, pp. 17–26.
4. I. Joe and S.C. Lee, "Bootup Time Improvement for Embedded Linux Using Snapshot Images Created on Boot Time," *Proc 2nd IEEE Int'l Conf. Next Generation IT* (ICNGIT 11), 2011, pp. 193–196.
5. "Fast Bootup of Froyo Edition of Android on Pathpartner Media Phone"; www.youtube.com/watch?v=TzAIuICGqh4.

and device state, which are stored on disk or flash memory. Because the system image is saved only once, startup is much faster—half that of the traditional SR method.

Our technique also uses a Linux kernel thread, `RSS_thread`, to synchronize system state. Because the same image is used for each boot, the new files on the external disk might not be synchronized with the system state saved in the image. To solve that problem, `RSS_thread` synchronizes the RAM image state with the more current files in external memory. In an evaluation of 30 tests, startup time with the FBFS method was consistently shorter than the optimized SR method, and the snapshot system state was accurately synchronized with the state in external memory.

SNAPSHOT IMAGING

Figure 1 shows snapshot imaging in traditional SR and with the FBFS technique.

After the Android system initially boots and displays the home screen, it creates a snapshot image of that screen. With our FBFS technique, the image is created once and then used repeatedly each time the user starts the device. The image might need to be re-created after a major software update, or if a saved image is corrupted, but these exceptions are rare.

Because Android OS is large and complex, the flash memory's (embedded multimedia card [eMMC] or secure digital [SD] card) reads and writes are relatively slow, so having the smallest possible snapshot image is important. To shrink the snapshot image during its creation, the FBFS technique uses the `echo > /sys/power/tuxonice/do_hibernate` command, which uses the least-recently used (LRU) page-replacement algorithm to move inactive memory pages to the disk's swap partition. Consequently, there is no need to save the inactive pages in the system image, which can reduce image size by 50 to 70 percent.

After the system is powered on, the boot loader and Linux kernel are loaded, and the kernel loads the system state from the saved snapshot image into RAM.

RESYNCHRONIZING SYSTEM STATE

Because the same image is used regardless of bootup speed, FBFS has the same synchronization problem as traditional methods that use a snapshot image. If

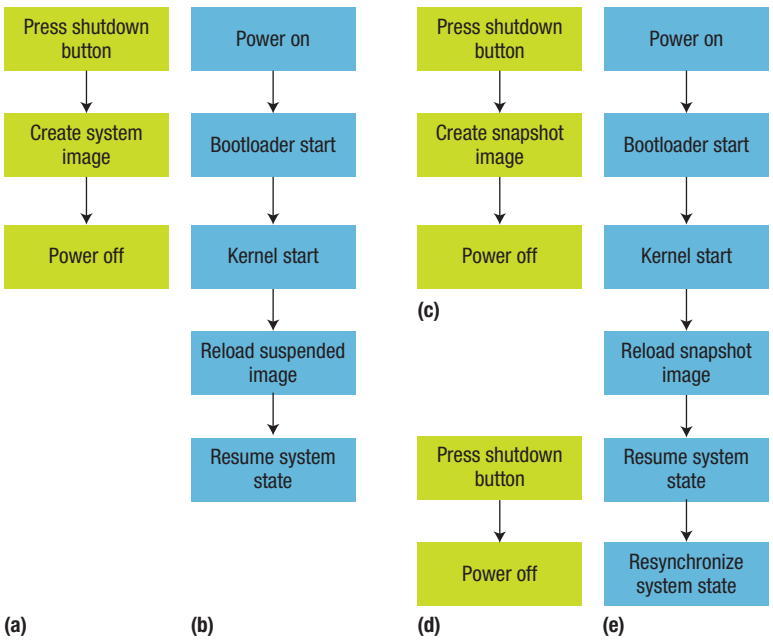

FIGURE 1. Procedural diagrams of the shutdown and startup for the suspend–resume (SR) technique and our fast boot, fast shutdown (FBFS) technique. (a) In the SR technique, for each shutdown, the OS must create a full system image and then suspend it. (b) At each startup, it must then reload the suspended image and the system state in that image, which will not be synchronized with the current system state if any downloads have taken place. (c) With the FBFS technique, the system creates a snapshot image only once during the first shutdown. (d) Other shutdown processes are the same as those of the normal Android OS, so the shutdown time is much shorter than with the SR technique. (e) At startup, the FBFS method resynchronizes system state instead of just resuming it.

the system is read only, synchronization is unnecessary, but consumer devices require system-software updates, and their users want the latest applications. Both automatic downloads and user operations modify system properties and user space during runtime, which affects system state and external memory. The resume operation restores only the state saved in the snapshot image stored in RAM, which could easily be out of synch with the physical files in external memory.

The FBFS method synchronizes the RAM image state with the up-to-date files in the eMMC by updating data structures in the Linux file system and services and applications in the Android platform layer.

To synchronize system state, the FBFS methods uses `RSS_thread`, a background kernel thread, which is set before the system is suspended in a regular boot and awakened after the snapshot image in RAM is reloaded. It first synchronizes the ext4 file system, and then goes on to synchronize the virtual file system (VFS), Android services, and Android applications. Figure 2 shows details of the synchronization process.

Synchronizing the ext4 file system

When the user or system accesses a file (regular files, directories, symbolic links, hard links, and so on), the Linux kernel will check whether the file is cached in RAM. If it does not find the file, the kernel will load it from external memory into RAM. To access files quickly, the Linux kernel will preserve the files' attributes and contents in RAM to the degree possible. As long as RAM files are up to date, the kernel will access them from RAM rather than from external memory.

Android 5.0 uses the ext4 file system, which implements a circular buffer that logs changes to the file system. After a crash, the file system can rapidly recover to a consistent state by just reading the contents from the journal buffer, thereby avoiding a more intrusive file-system scan and reducing time due to recovery.

`RSS_thread` checks if the RAM state in the image is synchronized with the up-to-date files in the eMMC partitions that house the user's data. It first scans the ext4 file system's journal system and then updates information on metadata and data use. If the scanned metadata was used, `RSS_thread` must find which operation made this change. If it was a deletion, the metadata and its data must be removed; if it was a modification, the attribute information for the data must be updated; and if was an addition, the data and metadata must be moved to RAM.

Synchronizing the virtual file system

To synchronize the VFS index nodes (inodes) and directory entries (dentries), RSS_thread updates them to their related up-to-date objects.

The nature of updating depends on the operation; Figure 2 shows the three most prevalent ones:

- *Deletion*. RSS_thread closes the file descriptor associated with the deleted file in the process's open file table. It then releases all resources of those files in memory including the inode, dentry, and data cache.
- *Modification*. Through the i_mapping field, RSS_thread accesses the most up-to-date flag of the inode structure's pages and then clears it. When users or applications try to access the contents of these files, the system will see that the pages are invalid and load the latest data instead. If the modification was to an attribute modification, the system will synchronize inodes and dentries with the latest state of inode objects in the objects bucket.
- *Addition*. RSS_thread creates new inode, dentry, and file descriptors for the files created during the last runtime.

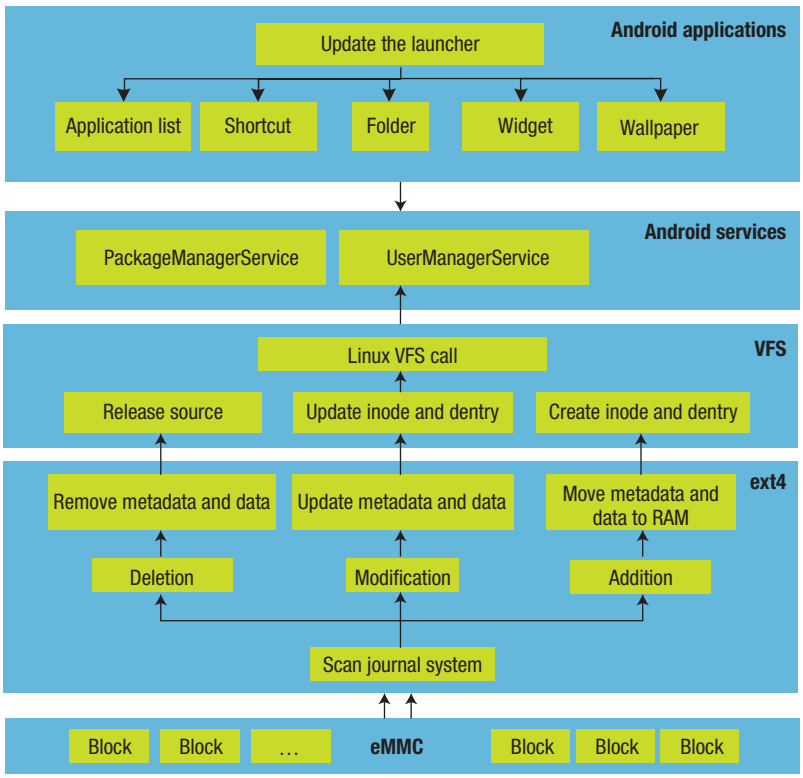

FIGURE 2. Synchronizing system state. The FBFS technique uses a background Linux kernel thread to synchronize the ext4 file system and virtual file system (VFS) with the latest files in the external multimedia card (eMMC). The thread then issues a broadcast intent for all Android applications and services to synchronize system states in the Android platform layer and, as a last step, updates the launcher application.

Synchronizing Android services

After updating structures in the Linux VFS and ext4 with the latest eMMC files, RSS_thread generates a broadcast intent Android.intent.action.SYNC_SYSTEM_AFTER_FAST_BOOT. Activities or services that receive this intent will perform their respective operations to synchronize system states in the Android platform layer. Two Android services require updating: PackageManagerService (PKMS) and UserManagerService.

PackageManagerService. PKMS manages the installation, uninstalling, and updating of system and user applications, and provides a channel for application-information queries. After receiving the Android.intent.action.SYNC_SYSTEM_AFTER_FAST_BOOT intent, PKMS will get the names of applications that have been installed, uninstalled, or updated since the boot image was created. For newly installed or updated applications, PKMS parses the corresponding Android application package (APK) files to get application information and saves the information into related structures, which allows the new applications to become available. For uninstalled applications, PKMS removes the related information saved in the PKMS space. When it completes

TABLE 1. Average startup and shutdown times across 30 tests.

Condition	Image size (Mbytes)	Startup time (s)	Shutdown time (s)
Normal boot	N/A	40.0	10.3
Traditional suspend–resume method	408	12.5	20.6
Fast boot, fast shutdown (FBFS) technique	202	7.8	10.3

a task, PKMS sends a broadcast intent related to the operations to inform other services of these changes.

UserManagerService. From Android OS 4.2 on, the system has provided multiuser support, so the UserManagerService must be updated by first using the UserManager interface to create or remove users, get user information, and get user interface handles. If a new user is added or an existing user is removed after the system restart, the users' information is no longer synched with the snapshot image. To address this problem, the FBFS technique removes old user information in the snapshot image and restarts UserManagerService to update user information. Updating occurs in three steps:

- get the list of user information stored in `ArrayList`,
- prune any partially created or partially removed users, and
- remove all information (including package manager settings, user list, and user file) of the pruned users.

Synchronizing Android applications

After synchronizing Android services, the FBFS technique must update the launcher—the application that users interact with most frequently and thus the one whose application properties have probably changed since snapshot creation. Updates include the application list, workspace, and the Settings app.

Application list. Operations such as install, uninstall, and update can change the application list's state. When these operations are performed, the launcher updates its application list and notifies other processes or services. At the same time, the Android system renews the packages.list and packages.xml files.

When the launcher receives the intent from PKMS, it extracts the application information and the operation type. For newly installed applications, it adds application information in `added_pkg`. For updated applications, it first removes the obsolete information in the application list and then adds new application information in `updated_pkg`. For uninstalled applications, the launcher adds application information to `uninstalled_pkg`.

Once it has updated the application list, the launcher simply refreshes the list in the screens according to the data saved in `added_pkg`, `updated_pkg`, and `uninstalled_pkg`.

Update workspace. The next step in application synchronization is to update shortcuts, folders, app widgets, and wallpapers in the workspace. The launcher reloads the launcher.db file, which contains all workspace-related information about the items, such as shortcuts and folders, needed to refresh the workspace with an updated view. It then parses the appwidgets.xml file and informs the App Widget Service (AWS) of widget changes. AWS sends its intent to hosts of these widgets, resulting in an update of all widgets in all hosts. Finally, the launcher loads concrete information about the wallpaper from the wallpaper_info.xml file. The new wallpaper is displayed and the WallPaperManager service sends out a broadcast intent to notify other services and applications of this change.

Update settings. The Settings app has an interactive interface that users interact with to change local attributes, including those for the network, system mode, and device. When users change an attribute, the system writes the changes into the settings.db file. Launching the settings after a restore clears cache contents and reloads the information from the file settings.db file.

Once Settings is updated, the FBFS technique freezes the `RSS_thread` because the Android system should be completely resynchronized at this point.

EVALUATION

To validate our FBFS method, we conducted 30 tests to evaluate boot and shutdown times, as well as system synchronization relative to normal booting and a traditional SR technique. We used a Google Nexus 4 as the testing hardware platform, which has a quad-core, 1,500-MHz CPU and a 2-Gbyte memory, and a built-in storage eMMC to store the

TABLE 2. Test results for raw Android and Monkey.

Category	No. of events	Restarts	No. of times application did not respond	No. of Crashes	Monkey runtime (h)
1	20,000	0	0	1	0.04
2	40,000	1	1	–1	–0.02
3	60,000	1	–1	2	–0.02
4	80,000	0	2	0	0.02
5	100,000	2	1	–1	–0.01

The last four columns report exception cases relative to test data in raw Android.

suspend image; the eMMC's read speed is 50 Mbytes per second (MBps) and write speed is 22.9 Mbps. Because synchronization modified the Linux kernel and Android system, we also conducted tests to verify whether the synchronized system was also stable.

Although our approach is not restricted to a particular Android version, our software environment is based on Android 5.0 and Linux kernel 3.4.0. All tests used the same system state and the same hardware.

Boot and shutdown time

Table 1 shows the average startup and shutdown times using a 408-Mbyte snapshot image, which the FBFS method reduced to 202 Mbytes. We defined startup as the time from power on to the display of the home screen, and shutdown time as the time the user presses the power-off button to when the device is fully off. The average startup time with our method is approximately 7.8 s, which includes the average U-boot and Linux kernel boot time (3 s), the average image-loading time (4 s), and the average synchronization time (0.8 s). The image-loading time depends on the external memory's read speed, and the synchronization time is dictated by the number of changes due to user operations and file changes.

The shutdown time is the same as normal shutdown (10.3 s), as it uses the same procedure except when it creates a saved image. Our 7.8-s startup time contrasts sharply with the 40-s startup time for a normal boot and is faster than the 12.5-s startup time for the traditional SR method. Our method also has cut shutdown time in half relative to the traditional SR method (10.3 s versus 20.6 s).

These results translate to an 80.5 percent reduction in normal boot time. The smaller image used in our method is 49.5 percent of the image size with the traditional SR method.

System synchronization

Tests for system synchronization evaluated the synchronization of file systems and Android services and applications.

File system synchronization

To verify that file systems were correctly synchronized, we booted the device and then performed operations on various files, directories, and links, including deletion, renaming, and content modification. We then rebooted the device and resumed system state from the same snapshot image. Restoring the snapshot image woke up `RSS_thread`, which synchronized memory state with the eMMC's physical files. We then checked all modified files and directories to see if their states reflected the changes we made. In all cases, the files and directories were accurately synchronized.

To test content modification, for example, we created the test.c file to verify synchronization accuracy. We deleted the file's first four lines, saved it, rebooted and resumed the system from the same snapshot image, and examined the file. Our test found that synchronization correctly changed the file, which was evidence that `RSS_thread` had worked and accurately synchronized the eMMC's data with data in RAM. We performed similar tests for all other operations with the same results.

Android services and applications synchronization

We also evaluated the synchronization of operations that manage Android applications and services (install, delete, and update) and special system applications (application list, workspace, and settings, such as wallpaper).

ABOUT THE AUTHORS

XIA YANG is an associate professor in the School of Information and Software Engineering at the University of Electronic Science and Technology of China. Her research interests include embedded systems, embedded OS security, formal methods, and software engineering. Yang received a PhD in computer science from the University of Electronic Science and Technology of China. She is a member of the China Computer Federation. Contact her at xyang@uestc.edu.cn.

PENG SHI is an MS student in the School of Information and Software Engineering at the University of Electronic Science and Technology of China. His research interests include embedded systems and fast-boot technology for Android. Shi received a BS in aircraft manufacturing and engineering from the Nanjing University of Aeronautics and Astronautics. Contact him at 996647604@qq.com.

HAIYONG SUN is a PhD student in the School of Information and Software Engineering at the University of Electronic Science and Technology of China. His research interests include embedded systems and formal methods. Sun received an MS in software engineering from the University of Electronic Science and Technology of China. Contact him at 1037792257@qq.com.

WENXUAN ZHENG is an MS student in the School of Computer Science and Engineering at the University of Electronic Science and Technology of China. Zheng received a BS in network engineering form Henan Normal University. Contact him at 744039342@qq.com.

JAMES ALVES-FOSS is a professor in the Department of Computer Science at the University of Idaho and director of the university's Center for Secure and Dependable Systems. His research interests include the design and analysis of secure distributed systems, with a focus on formal methods and software engineering. Alves-Foss received a PhD in computer science from the University of California, Davis. He is a Senior Member of IEEE and ACM. Contact him at jimaf@uidaho.edu.

System stability

We tested system stability by adapting the FBFS technique to be compatible with Android's Monkey test tool. We used Monkey to generate random user events. When the events completed, we analyzed the response of the tested system. Events were in one of five categories; for each category, we repeated the test 100 times, and each test consisted of multiple events. Table 2 shows the number of events in each category, along with the results for our approach and for raw Android as the average number of times we found exception cases relative to raw Android for each test. We were interested in stability, so our focus was on the average exception-case changes relative to raw Android. As the minimal change range (–1 to 2) implies, our approach does not undermine system stability.

Android OS is becoming more widespread, but its complexity and size cause startup and shutdown delays that make consumers impatient and lower their willingness to invest in Android devices. The FBFS technique reduces these times through the use of a much smaller snapshot image that is created only once. The technique is also suitable for devices that are not battery-powered, because there is no need to create an image for every shutdown. In contrast, the SR method requires power, possibly from a battery, to create the image during shutdown.

Tests to evaluate startup and shutdown times, synchronization, and stability showed that the FBFS technique works well. Resynchronization takes about 1 s, but using a single image enabled an average shutdown time of 10.3 s, half of the 20.6-s shutdown time with a traditional SR method. Additional work is needed to determine if resynchronization time will increase over time, which would require setting a threshold synchronization time that will trigger the creation of a new standard boot image. Such a threshold would be valuable when a major OS update is released, for example, or to shorten the startup times for large embedded systems, such as digital TVs.

Although our study was conducted on Android 5.0, we are confident that the FBFS method will easily scale to Android 6, even with its new power-management modes. The FBFS technique performs independently of these modes, so users should still see similar performance improvements.

REFERENCES

1. J.H. Kim, H. Roh, and J. Lee, "Improving the Startup Time of Digital TV," *IEEE Trans. Consumer Electronics*, vol. 55, no. 2, 2009, pp. 721–727.
2. G. Singh, K. Bipin, and R. Dhawan, "Optimizing the Boot Time of Android on Embedded Systems, *Proc. 15th Int'l IEEE Symp. Consumer Electronics* (ISCE 11), 2011, pp. 503–508.
3. X. Yang, N. Sang, and J. Alves-Foss, "Shortening the Boot Time of Android OS," *Computer*, vol. 47, no.7, 2014, pp. 53–58.