2018 6th IEEE International Conference on Mobile Cloud Computing, Services, and Engineering

Device-Specific Linux Kernel Optimization for Android Smartphones

Pengfei Yuan, Yao Guo, Xiangqun Chen, and Hong Mei
Key Laboratory of High-Confidence Software Technologies (Ministry of Education)
School of Electronics Engineering and Computer Science, Peking University, Beijing, China, 100871
Email: {yuanpf12, yaoguo, cherry, meih}@sei.pku.edu.cn

***Abstract*—To make smartphones more powerful, researchers have proposed many techniques to improve the performance of mobile systems and applications. As the most popular mobile operating system, Android is based on the Linux kernel. Therefore optimizing kernel performance can potentially accelerate Android smartphones.**

In this paper, we propose a compiler-based approach to constructing device-specific optimized Linux kernels for Android smartphones. By utilizing runtime feedback from the device, we can instruct the compiler to perform profile-guided optimization (PGO) and produce a Linux kernel image optimized specifically for the device, which can be shipped together with the device when it is manufactured, or released later in an update of the whole system. To the best of our knowledge, this paper is the first work that successfully applies PGO to the Linux kernel on Android smartphones to gain performance improvement.

We conduct experiments on three different Android smartphones, namely Nexus 5X, Nexus 6 and Nexus 5. Results show that system performance improves by 11.1%, 4.1% and 9.3% on average, respectively. Specifically, we measure performance improvements of key Android system components such as multithreading and task scheduling, Binder, and storage and file systems.

***Index Terms*—Android operating system, Linux kernel, performance, profile-guided optimization.**

1. Introduction

The rapid spread of smartphones makes mobile computing prosperous. Due to device size and battery constraints, the computational power of smartphones is still limited when compared to desktops and servers. To support more sophisticated applications, mobile cloud computing (MCC) techniques such as COMET [1] and Uniport [2] have been proposed to incorporate the more powerful cloud computing infrastructure into mobile devices. On the other hand, many research work have explored performance optimizations for mobile systems and applications. For example, F2FS [3] improves file system performance for flash storage. SmartIO [4] reduces `iowait` delay on smartphones. PerfChecker [5] detects performance bugs in smartphone applications.

As the mobile operating system (OS) which has the highest market share, Android is based on the Linux kernel. Since kernel performance is critical to the efficiency of the whole system, optimizing performance for the Linux kernel can accelerate Android apps running on top of it. In this paper, we adopt a compiler-based approach, namely profile-guided optimization (PGO), to construct device-specific optimized Linux kernels for Android mobile devices. Using the optimized kernel, we can improve performance for critical Android system components such as multithreading and task scheduling, Binder inter-process communication (IPC), and storage and file system.

Our previous work [6] has demonstrated that using the PGO technique to optimize the Linux kernel for a specific application is feasible on x86-based machines. In this paper, we successfully apply the PGO technique to the Linux kernel on Android smartphones. Unlike application-specific kernel optimization performed on x86-based machines, we aim to achieve *device-specific kernel optimization* for Android smartphones.

The device-specific optimized Linux kernel can be optimized for each new or existing device. For a new device, it can be shipped together with the device when it is manufactured. For an existing device, it can be installed during a system update. By running an instrumented Linux kernel on the mobile device and collecting appropriate profile feedback, we can instruct the compiler, namely GCC, to perform advanced optimizations and generate a Linux kernel image optimized specifically for a particular device.

We make the following main contributions in this paper:

- We propose a compiler-based approach to constructing device-specific optimized Linux kernels for Android smartphones. During the actual optimization process, users do not need to make any manual modifications to the Linux kernel.
- We have implemented the proposed optimization and selected benchmarks for Android kernel performance measurements. Based on our experiments on three different Android smartphones, compared to the default `-O2` optimization, our approach improves system performance by 4.1–11.1%.
- To the best of our knowledge, this paper is the first work that successfully applies the PGO technique to

2573-7562/18/$31.00 ©2018 IEEE
DOI 10.1109/MobileCloud.2018.00018

the Linux kernel on Android smartphones to gain performance speedup. Since it is a compiler-based approach, other Android system improvements can employ this technique to achieve additional performance speedup.

The rest of this paper is organized as follows. Section 2 gives the background knowledge on PGO and GCC optimizations for the Linux kernel. Section 3 presents the detailed design and implementation of our proposed approach in device-specific optimization of the Linux kernel for Android smartphones. Section 4 presents benchmarks and experimental environments, as well as the experimental results on different smartphones. Section 5 discusses related work, while Section 6 concludes our work.

2. Background

2.1. Profile-Guided Optimization

Profile-guided optimization (PGO) has been well studied in the compiler community [7]. By using feedback information such as control flow graph and expression value profiles, which are collected in one or more previous program runs, the compiler focuses its optimization efforts on the frequently executed portions of the program. PGO has been applied to large open source projects such as Firefox [8], Chrome [9] and GCC itself [10], and achieved 5–17% speedup.

A typical PGO process consists of the following phases:

- **Instrumentation**. The compiler instruments the target program during compilation in order to collect profile information that will be used for later optimizations. The profile information consists of control flow traces, value and address profiles, etc.
- **Profile collection**. The instrumented target program is executed to collect profile information. The execution process should reflect real-world runtime scenarios.
- **Optimization**. The compiler uses the profile information collected in the previous phase to optimize the target program. The profile information helps the compiler make better decisions on branch prediction, basic block reordering, function inlining, loop unrolling, etc.

Recent GCC versions support sampling-based AutoFDO [11], which does not need program instrumentation. It requires last branch record support from Intel processors. However, most Android smartphones use ARM processors. So we do not adopt AutoFDO in this paper.

2.2. PGO in the Linux Kernel

We have shown that PGO is applicable to the Linux kernel on x86-based machines (such as desktops and servers) in previous work [6], the goal of which is to construct application-specific optimized Linux kernels. By collecting feedback information from the instrumented Linux kernel when running the target application, and using the feedback information in compiler optimization of the kernel, we can build an optimized kernel image which provides relatively optimal performance for the target application.

To enable the PGO technique in the Linux kernel, we have overcome the following challenges:

- **Enabling kernel instrumentation**. We have added kernel instrumentation support to the Linux kernel and GCC, and enabled the instrumented kernel to boot on x86-based machines.
- **Collecting kernel profile feedback**. We have implemented tools to collect and process kernel profile feedback. By specifying appropriate start and end of profile collection, we can collect kernel feedback information that is application-specific.
- **Choosing correct optimization**. We have chosen compiler optimization options that are suitable for kernel optimization, which can improve performance, reduce code size, and ensure correctness.

3. Design and Implementation

3.1. Challenges

While using PGO in user applications is as simple as enabling a few compiler options and applying PGO to the Linux kernel on x86-based machines is implemented in our previous work, adopting PGO to construct device-specific optimized kernels for Android smartphones faces the following new technical challenges:

- **Kernel instrumentation**. Android smartphones generally use legacy Linux versions. Therefore, we need to backport kernel instrumentation support and add support for ARM and ARM64 processors, which are used in Android smartphones.
- **Booting the instrumented kernel**. Since the booting procedure of Android devices is complex and the Android bootloader is proprietary, we need to perform black-box testing and reverse engineering to solve booting issues of the instrumented kernel.
- **Device-specific optimization**. To achieve device-specific kernel optimization, we should first collect device-specific kernel feedback information. Then we should choose compiler optimization options that are suitable for optimizing the Linux kernels of Android smartphones.

3.2. Design Overview

In previous work [6], we have successfully applied PGO to the Linux kernel on x86-based machines. To apply PGO to the Linux kernel on ARM/ARM64-based Android

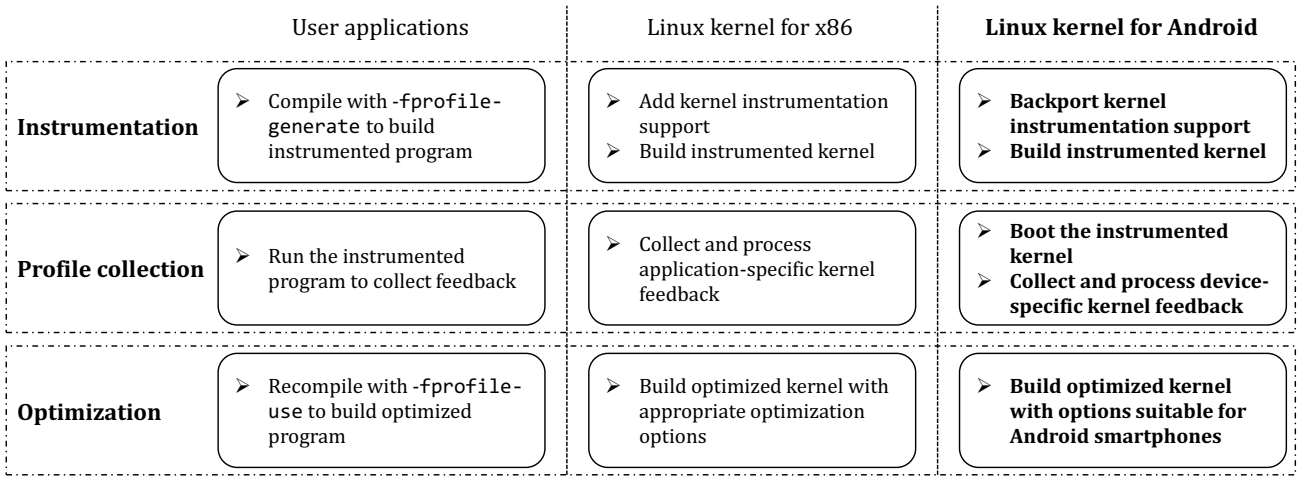

Figure 1. Comparison of applying PGO to user applications, the Linux kernel on x86-based machines and ARM/ARM64-based Android smartphones.

smartphones, we propose an approach illustrated in Figure 1, in comparison with applying PGO to user applications and the kernel for x86 machines.

For kernel instrumentation on Android smartphones, we need to backport relevant kernel modifications to legacy Linux versions and build instrumented kernel images for different devices. For kernel profile feedback collection, we need to boot the instrumented kernel on the device and use appropriate tools to collect and process device-specific feedback information. For kernel optimization, we need to choose compiler optimization options suitable for the Linux kernel of the Android system.

3.3. Kernel Instrumentation

Backporting of kernel instrumentation support to legacy Linux versions consists of the following aspects:

- **Update of the Linux *gcov* subsystem**. The *gcov* subsystem provides support of control flow instrumentation, which is required by the PGO technique in GCC. However, legacy Linux versions only support GCC 4.6, which was released in 2011, and earlier versions. We backport related kernel patches [12], [13] and update the *gcov* subsystem to support the latest GCC versions.
- **Support of ARM and ARM64 architectures**. We modify the kernel build system to support instrumenting the kernel on ARM and ARM64 architectures, which are typically used in Android smartphones.
- **Support of data flow instrumentation**. Besides control flow instrumentation, the PGO technique in GCC also requires data flow instrumentation of the target program, namely the Linux kernel. We have discussed this part of implementation in our previous work [6].

TABLE 1. SUMMARY OF BACKPORTING KERNEL INSTRUMENTATION SUPPORT FOR ANDROID SMARTPHONES.

Category	Lines of code
Update of the Linux *gcov* subsystem	907
Support of ARM and ARM64 architectures	17
Support of data flow instrumentation	999
Fixing compilation errors	62
Total	1,985

- **Fixing compilation errors**. Since the Linux versions of Android smartphones are typically older, there are compilation errors when building the kernel. For ARM, we may encounter errors about the `inline` keyword. For ARM64, we may encounter errors about GCC's built-in functions. We modify the kernel source code to fix these compilation errors.

As a summary of our backporting work, Table 1 shows the lines of kernel source code we have modified.

3.4. Booting the Instrumented Kernel for Profile Collection

To collect kernel feedback information from Android smartphones, we must first boot and run the instrumented kernel. The booting process of ARM/ARM64-based Android devices can be summarized as follows [14]:

1) The processor powers on and executes the primary bootloader stored in BootROM.
2) Since the space in BootROM is limited, a secondary bootloader stored in eMMC Flash is loaded and executed, which initializes ARM TrustZone [15].
3) The actual Android bootloader is loaded and executed, which provides the *fastboot* [16] feature.

TABLE 2. ANDROID BOOTING PARAMETERS.

Name	Description
`base`	Base address of physical memory
`kernel-offset`	Offset of kernel image
`tags-offset`	Offset of kernel parameters
`ramdisk-offset`	Offset of init ramdisk
`page-size`	Page size of the eMMC Flash storage

4) The Linux kernel image and init ramdisk are loaded and the Android system starts up.

To ensure secure booting, each step verifies the digital signature of the loaded content in the next step [17]. In order to boot the instrumented kernel, we need to deal with the Android bootloader, which is responsible for loading the kernel image. By executing the "fastboot oem unlock" command, we can unlock the Android bootloader and disable its signature verification. Then we can boot customized kernel images on the device.

We can use the "fastboot boot" command to boot customized kernel images. Table 2 lists the parameters we must set when using that command. We can get the default values of these parameters by extracting the official *boot.img* file.

However, the instrumented kernel usually fails to boot with the default parameters, because its image size is much larger than normal kernel images compiled with `-O2` or `-Os`. We need to adjust the parameters and determine their appropriate values through trial and error.

To assist the black-box testing process, we can perform reverse engineering analysis on the Android bootloader, which can be extracted from the *aboot* partition of the device using the *dd* command [14]. To simplify reverse engineering analysis, we can refer to the source code of LittleKernel [18], which most Android bootloaders are based on.

For some devices, the parameter values are built into the Android bootloader and cannot be specified via command-line. In this case, we need to reduce the image size of the instrumented kernel so as to boot it on the device. Specifically, we adopt selective kernel instrumentation. By instrumenting only the key components of the Linux kernel, such as task scheduler, memory management, file system and Binder, we can greatly reduce the image size.

After solving the booting issues, we can run the instrumented kernel on the device and collect feedback information. The profile collection tool is implemented as a shell script. To execute the script on Android smartphones, we employ the *busybox-static* [19] package from Debian, which provides statically-linked ARM and ARM64 binaries that can directly run on Android smartphones and support the execution of shell scripts.

To collect device-specific profile feedback for kernel optimization, we need to run a set of Android apps that can reflect real-world runtime scenarios on top of the instrumented kernel. Specifically, we can choose the most popular apps from Google Play or other local app stores.

3.5. Device-Specific Optimizations

For devices that use selective kernel instrumentation, the collected feedback information is incomplete. For kernel source code that the feedback information does not cover, we need to disable optimization options that only benefit if there is profile feedback. Specifically, we can replace the `-fprofile-use` option with `-fbranch-probabilities` [20].

Our compiler-based approach of kernel optimization is device-specific in the following aspects:

- Different devices use different Linux versions and build configurations.
- Different devices may use different GCC versions for kernel optimization.
- The feedback information collected during profiling is device-specific. These feedback information will be used in profile-guided kernel optimization.
- Detailed compiler options used in kernel optimization is device-specific.

4. Evaluation

4.1. Benchmarks

To evaluate system performance of Android smartphones, we should use benchmarks that are system intensive, which invoke OS features intensively. Popular benchmarks like GeekBench and AnTuTu are computational intensive, thus unsuitable for our evaluation. Therefore, we have implemented and collected a set of benchmarks suitable for Android kernel performance measurement, which measure performance of key Android system components such as multithreading and task scheduling, Binder, and storage and file system.

Previous work [21] has shown that correctly benchmarking performance of Android smartphones is challenging. To ensure that the performance results are valid and stable, we set the CPU scaling governor to *performance*, use the *taskset* command to set the CPU affinity of the benchmarks, and stop performance throttling services like *thermal-engine* and *mpdecision*.

4.1.1. Multithreading and Task Scheduling. Android apps generally use multiple threads. The main thread is responsible for user interface (UI) and interaction. Executing time-consuming tasks in the main thread will freeze the whole application. Therefore, Android apps make use of APIs like `AsyncTask`, `HandlerThread` and `ThreadPoolExecutor` to execute time-consuming tasks in background threads.

To measure multithreading and task scheduling performance of the Android system, we implement a benchmark application, which repeatedly executes the following operations and measures throughput:

1) Creating a set of `java.lang.Thread` instances according to the benchmarking concurrency.

2) Invoking the `start` method of the `Thread` instances.
3) Invoking the `join` method of the `Thread` instances to wait for completion.

During its lifetime, each created thread executes the following operations:

- Invoking the `setThreadPriority` method of class `android.os.Process` to set the scheduling priority of the current thread to the background level.
- Invoking the `Thread.yield` method a few times to explicitly release CPU and trigger task scheduling in the kernel.

4.1.2. Binder IPC. Binder is a key component of the Android system. It is used for nearly everything that happens across processes in the core platform [22]. System services like `ActivityManger`, `WindowManger`, `PackageManager` and inter-component communication in Android apps all use Binder IPC, which is supported by a kernel module.

To measure Binder IPC performance, we implement two benchmark apps. One runs a background service, which is based on class `android.app.IntentService` and declared in *AndroidManifest.xml*. The other runs in foreground to show benchmarking results. The two apps communicate with each other via Binder IPC. The performance measurement procedure is as follows:

1) The foreground application responds to the `onClick` event of its UI button. It puts the `android.os.Messenger` instance of the current process in `android.content.Intent`, and invokes the `startService` method of class `android.content.Context` to start the background service.
2) The background service responds to the `onHandleIntent` event of class `IntentService` and sends the `Messenger` instance of the current process to the foreground application via invoking the `send` method of the foreground application's `Messenger` instance.
3) The foreground application responds to the `handleMessage` event of class `android.os.Handler` which binds to its `Messenger` instance, and communicates with the background service with the latter's `Messenger` instance. It sends multiple messages simultaneously according to the benchmarking concurrency.
4) The background service responds to the `handleMessage` event of class `Handler` which binds to its `Messenger` instance, and communicates with the foreground application with the latter's `Messenger` instance.
5) The previous two steps are repeatedly executed. The foreground application is responsible for calculating Binder IPC throughput and showing the result.

4.1.3. Storage and File System. Storage and file system is a very important OS component. To measure its performance, we choose SQLite, which is an embedded database widely used in Android apps, as the benchmark. The application we use is *speedtest1* [23], which is an official benchmark application of SQLite. We cross-compile it to ARM and ARM64 versions so as to run it on Android smartphones. The test load parameter `--size` is set to 20 and the test database file is put in the *data* partition.

4.2. Devices and Experimental Setup

The Android smartphones used in our experiments are Nexus 5X, Nexus 6 and Nexus 5, which are officially released by Google. Table 3 lists their specifications.

The Qualcomm Snapdragon 808 processor used in Nexus 5X adopts the ARMv8-A 64-bit instruction set architecture. The processor has six cores, two of which are based on Cortex-A57 with 1.82GHz peak frequency, four of which are based on Cortex-A53 with 1.44GHz peak frequency. The compiler we use for Nexus 5X is GCC 6.4 ARM64 cross-compiler. Since the frequencies of Cortex-A57 and Cortex-A53 cores differ significantly, we specify the benchmarks to run on the same type of cores when setting CPU affinity with the *taskset* command.

The Qualcomm Snapdragon 805 processor used in Nexus 6 adopts the ARMv7-A 32-bit instruction set architecture. The processor has four Krait 450 cores with 2.7GHz peak frequency. The compiler we use for Nexus 6 is GCC 5.4 ARM cross-compiler because the Linux kernel compiled with GCC 6.4 and `-O3` does not boot on Nexus 6.

The Qualcomm Snapdragon 800 processor used in Nexus 5 also adopts the ARMv7-A 32-bit instruction set architecture. It has four Krait 400 cores with 2.26GHz peak frequency. The compiler we use for Nexus 5 is also GCC 5.4 ARM cross-compiler because the Linux kernel compiled with GCC 6.4 and `-Os` does not boot on Nexus 5.

The three phones run LineageOS 14.1 [24], which is based on Android 7.1.2 and a continuation of the famous CyanogenMod project. To install LineageOS, we unlock bootloaders for the phones and flash the TWRP recovery [25] into them. In order to facilitate performance evaluation, we install the *su* addon of LineageOS on the phones to acquire root permission.

The Linux kernel versions of the three phones are outdated. Linux 3.10 used in Nexus 5X and Nexus 6 is released in 2013 while Linux 3.4 used in Nexus 5 is released in 2012. This is a common issue for Android smartphones. Because some device drivers of the phones are closed-source binaries, we cannot manually upgrade the kernels. The kernel build configurations we used are the defaults provided by LineageOS, which are *lineageos_bullhead_defconfig*, *shamu_defconfig* and *lineageos_hammerhead_defconfig* for Nexus 5X, Nexus 6 and Nexus 5, correspondingly.

For Nexus 5X, we need to adjust booting parameters *kernel-offset*, *tags-offset* and *ramdisk-offset* to avoid overlapping of kernel image and init ramdisk in physical

TABLE 3. SPECIFICATIONS OF THE ANDROID SMARTPHONES USED FOR EVALUATION.

Device	Nexus 5X	Nexus 6	Nexus 5
Manufacturer	LG	Motorola	LG
Codename	Bullhead	Shamu	Hammerhead
Release date	Sep. 2015	Oct. 2014	Oct. 2013
Processor	Qualcomm Snapdragon 808	Qualcomm Snapdragon 805	Qualcomm Snapdragon 800
RAM	2GB LPDDR3	3GB LPDDR3	2GB LPDDR3
Storage	eMMC Flash		
Operating system	LineageOS 14.1		
Kernel version	3.10.73	3.10.40	3.4.0
File system	Ext4		

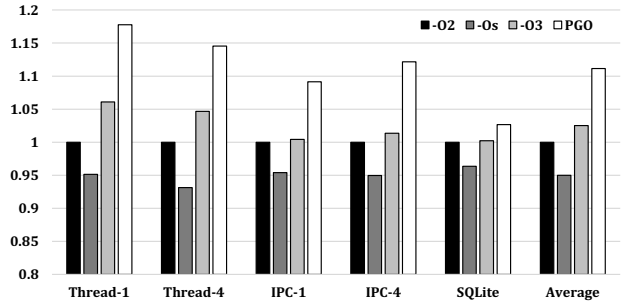

Figure 2. Performance comparison of different optimization configurations for Nexus 5X. (All numbers are are normalized to -O2. *Higher is better.*)

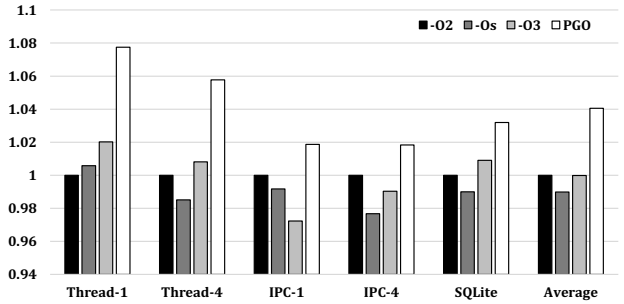

Figure 3. Performance comparison of different optimization configurations for Nexus 6. (All numbers are are normalized to -O2. *Higher is better.*)

memory space and boot the instrumented kernel on the phone.

For Nexus 6, we find that its booting parameters are built into the Android bootloader. Therefore, we use selective instrumentation to build the instrumented kernel and choose compiler options to build the optimized kernel accordingly.

For Nexus 5, we only need to increase the parameter values of *tags-offset* and *ramdisk-offset* to boot the instrumented kernel on the phone.

4.3. Results and Analysis

We have run different benchmarks on each phone with kernels compiled with different GCC optimization configurations, including `-O2`, `-Os`, `-O3` and PGO. Because `-O2` has been used as the default optimization configuration, we show relative performance numbers that are normalized to the performance results of `-O2`. All "average" numbers are calculated as geometric means.

4.3.1. Nexus 5X. Figure 2 shows the experimental results on Nexus 5X. In the figure, *Thread-1* and *Thread-4* represent the multithreading and task scheduling benchmark with concurrency set to 1 and 4, respectively. The same rule applies to *IPC-1* and *IPC-4* for the Binder IPC benchmark.

We can see that PGO greatly improves Binder IPC, multithreading and task scheduling performance. It also improves SQLite performance, which represents storage and file system performance. For the *Thread-1* benchmark, the speedup is over 17%. On average, which is calculated as geometric mean, system performance of Nexus 5X improves by 11.1%.

Besides PGO, we also compare performance results of kernel compiled with `-Os` and `-O3`. Compared with `-O2`, `-Os` degrades system performance on all benchmarks, while `-O3` improves performance. On average, the degradation of `-Os` is 5% and the improvement of `-O3` is 2.5%.

4.3.2. Nexus 6. Figure 3 shows the experimental results on Nexus 6. We can see that PGO greatly improves multithreading and task scheduling performance. It also improves Binder IPC and SQLite performance. On average, system performance of Nexus 6 improves by 4.1%. The speedup of Nexus 6 is substantially lower than Nexus 5X, because we employ selective instrumentation and optimization of the kernel on this device.

Compared with `-O2`, `-Os` and `-O3` improve performance for some benchmarks, while degrade performance for other benchmarks. It shows that all the three optimization options are suboptimal for Nexus 6.

4.3.3. Nexus 5. Figure 4 shows the experimental results on Nexus 5. We can see that PGO greatly improves Binder IPC, multithreading and task scheduling performance. It also improves SQLite performance. For the *Thread-4* benchmark, the speedup is nearly 20%. The speedup of SQLite is lower mainly because the I/O operations of the eMMC Flash storage are not accelerable. On average, system performance of Nexus 5 improves by 9.3%, which is comparable to Nexus 5X and much higher than Nexus 6.

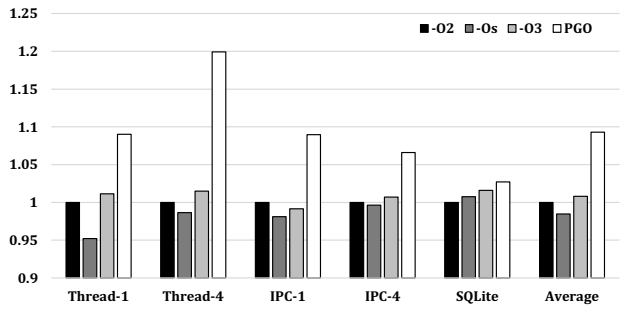

Figure 4. Performance comparison of different optimization configurations for Nexus 5. (All numbers are are normalized to -O2. *Higher is better.*)

Compared with `-O2`, `-Os` degrades system performance by 1.5% on average, while `-O3` improves performance by 0.8%. For specific benchmarks, different optimization options may perform better. However, PGO always generates the best performance.

4.4. Discussions

Compared to our previous work on application-specific Linux kernel optimization [6], this work have been able to achieve device-specific kernel optimization for mobile devices such as smartphones. Specifically, the proposed approach has the following benefits:

- The optimization is device-specific in the sense that we collect feedback information through instrumentation-based profiling while running a set of system-intensive applications on a specific device. Because the general principle of PGO is optimization based on profiles, thus the resulting kernel is optimized by the profiles collected from each device.
- The optimization itself is generic because the PGO technique itself is a relatively generic optimization mechanism. With application-specific profiles, it will generate application-specific optimized kernels. While with device-specific profiles, it can then generate device-specific optimized kernels.
- The optimization can be applied in a semi-automated process. We have constructed a tool chain to enable automated kernel patching and profiling, as well as automated GCC optimization. The only stage needing human intervention during the whole process is during the profile collection phase, when human needs to be involved in rebooting the device and running specific benchmarks.

Our work also faces some limitations:

- In order to generate kernels optimized for each device, we need to run appropriate benchmarks during the profile collection stage. However, there does not exist a suitable benchmark suite with Android apps. In fact, there are no widely-used Android benchmark suites of any kinds. We have chosen to implement and collect our own benchmark apps due to this reason. However, we cannot guarantee that the apps used are representative benchmarks. If we can find a better benchmark suite, it may potentially generate more representative profiles, and which in turn will help generate better optimized kernels.
- In order to perform PGO, the Linux kernel needs to be patched first. Although we have implemented automated kernel patching, some companies we have contacted are still concerned with how to maintain the consistency between the patched kernel and the mainline kernel. Although we have already attempted to incorporate our modification to the mainline Linux kernel, it will be a lengthy process that does not have any guarantees.
- The Linux kernel has been typically compiled using `-O2` and `-Os`. Although we have shown that it is possible to create a stable kernel with more aggressive optimizations such as using `-O3` [26], it has not been widely adopted and tested in real world. As PGO has applied some of the aggressive optimizations used by `-O3`, it is also one of the main concerns in whether it will generate a robust kernel. We will try to investigate this further in our future work.

5. Related Work

To augment the computational power of mobile devices, many research works study MCC techniques. MCC frameworks like Uniport [2] require modification and even reimplementation of mobile applications, while MCC systems like COMET [1] require extensive modification and implementation in the runtime environment such as the Dalvik virtual machine. Our compiler-based approach directly optimizes the OS kernel to improve system performance of mobile devices. We only need to replace the factory kernel image with the device-specific optimized kernel image. Mobile applications on the device will then have relatively optimal performance.

There are other research works that explore performance optimizations for mobile systems and applications. For example, F2FS [3] is a new file system that improves performance for Flash storage. SmartIO [4] is a system that reduces `iowait` delay on smartphones. PerfChecker [5] is a tool that detects performance bugs in smartphone applications. These works require modification and implementation in OS or applications, while our compiler-based approach does not need any source code modification when constructing the optimized kernel.

While our previous work [6] mainly focused on application-specific kernel optimization for x86-based machines, this paper focuses on device-specific kernel optimization for Android smartphones. We overcome several technical challenges to successfully apply the PGO technique to the Linux kernel of the Android system.

6. Conclusion

We have presented a compiler-based approach that takes advantage of profile-guided optimization (PGO) to construct device-specific optimized Linux kernels for Android smartphones. Specifically, we have solved problems with kernel instrumentation, booting of the instrumented kernel and device-specific kernel optimization. For performance evaluation, we have implemented and collected benchmarks for key Android system components such as multithreading and task scheduling, Binder, and storage and file system. Experimental results show that the average system performance speedups of Nexus 5X, Nexus 6 and Nexus 5 smartphones are 11.1%, 4.1% and 9.3%, respectively.

Acknowledgments

This work was partly supported by the National Key Research and Development Program (No. 2017YFB1001904) and the National Natural Science Foundation of China (No. 61772042). Yao Guo (email: yaoguo@pku.edu.cn) is the corresponding author.

References

[1] M. S. Gordon, D. A. Jamshidi, S. Mahlke, Z. M. Mao, and X. Chen, "COMET: Code offload by migrating execution transparently," in *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. Hollywood, CA: USENIX, 2012, pp. 93–106.

[2] P. Yuan, Y. Guo, and X. Chen, "Uniport: A uniform programming support framework for mobile cloud computing," in *2015 3rd IEEE International Conference on Mobile Cloud Computing, Services, and Engineering*, March 2015, pp. 71–80.

[3] C. Lee, D. Sim, J. Hwang, and S. Cho, "F2FS: A new file system for Flash storage," in *13th USENIX Conference on File and Storage Technologies (FAST 15)*, Santa Clara, CA, Feb. 2015, C, pp. 273–286.

[4] D. T. Nguyen, G. Zhou, G. Xing, X. Qi, Z. Hao, G. Peng, and Q. Yang, "Reducing smartphone application delay through read/write isolation," in *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '15, 2015, C, pp. 287–300.

[5] Y. Liu, C. Xu, and S.-C. Cheung, "Characterizing and detecting performance bugs for smartphone applications," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. ACM, 2014, pp. 1013–1024.

[6] P. Yuan, Y. Guo, and X. Chen, "Experiences in profile-guided operating system kernel optimization," in *Proceedings of 5th Asia-Pacific Workshop on Systems*, ser. APSys '14, 2014, C, pp. 4:1–4:6.

[7] R. Gupta, E. Mehofer, and Y. Zhang, "Profile guided code optimizations," in *The Compiler Design Handbook*, Y. N. Srikant and P. Shankar, Eds. CRC Press, 2002.

[8] T. Mielczarek, "Speed++." [Online]. Available: https://blog.mozilla.org/ted/2008/02/29/speed/

[9] S. Marchand, "Making Chrome on Windows faster with PGO." [Online]. Available: https://blog.chromium.org/2016/10/making-chrome-on-windows-faster-with-pgo.html

[10] Free Software Foundation, "Installing GCC: Building." [Online]. Available: https://gcc.gnu.org/install/build.html

[11] D. Chen, N. Vachharajani, R. Hundt, S.-w. Liao, V. Ramasamy, P. Yuan, W. Chen, and W. Zheng, "Taming hardware event samples for FDO compilation," in *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '10, 2010, C, pp. 42–52.

[12] F. Hrbata, "gcov: add support for gcc 4.7 gcov format." [Online]. Available: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=5f41ea0386a5

[13] ——, "kernel: add support for init_array constructors." [Online]. Available: https://lkml.org/lkml/2013/9/4/463

[14] J. Levin, "Reverse engineering Android's aboot." [Online]. Available: http://newandroidbook.com/Articles/aboot.html

[15] ARM Ltd., "ARM TrustZone." [Online]. Available: https://www.arm.com/products/security-on-arm/trustzone

[16] T. Bird, "Android fastboot." [Online]. Available: http://elinux.org/Android_Fastboot

[17] Android Open Source Project, "Verifying boot." [Online]. Available: https://source.android.com/security/verifiedboot/verified-boot.html

[18] T. Geiselbrecht, Google Inc., and The Linux Foundation, "LK bootloader." [Online]. Available: https://source.codeaurora.org/quic/la/kernel/lk/

[19] Debian Install System Team, B. Blank, and M. Tokarev, "Standalone rescue shell with tons of builtin utilities." [Online]. Available: https://packages.debian.org/sid/busybox-static

[20] Free Software Foundation, "Options that control optimization." [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

[21] Y. Guo, Y. Xu, and X. Chen, "Freeze it if you can: Challenges and future directions in benchmarking smartphone performance," in *Proceedings of the 18th International Workshop on Mobile Computing Systems and Applications*, ser. HotMobile '17. ACM, 2017, pp. 25–30.

[22] D. Hackborn, "Re: staging: android: binder: Remove some funny && usage." [Online]. Available: https://lkml.org/lkml/2009/6/25/3

[23] D. R. Hipp, "A program for performance testing." [Online]. Available: http://www.sqlite.org/src/artifact/7b1ab42b097b484c

[24] The LineageOS Project, "LineageOS Android distribution." [Online]. Available: https://lineageos.org/

[25] Team Win LLC, "TeamWin - TWRP." [Online]. Available: https://twrp.me/

[26] P. Yuan, Y. Guo, and X. Chen, "Rethinking compiler optimizations for the linux kernel: An explorative study," in *Proceedings of the 6th Asia-Pacific Workshop on Systems*, ser. APSys '15, 2015, C, pp. 2:1–2:7.