MEPOF: A Modular and End-to-End Profile-Guided Optimization Framework for Android Kernels

Keyuan Zong, Baojian Hua*, Yang Wang*, Shuang Hu, and Zhizhong Pan
School of Software Engineering, University of Science and Technology of China
Suzhou Institute for Advanced Research, University of Science and Technology of China
{zongky, guangan, sg513127}@mail.ustc.edu.cn, {bjhua, angyan}@ustc.edu.cn*

***Abstract*—Profile-Guided Optimization (PGO) is a novel compiler optimization leveraging runtime feedback and has been applied successfully to optimize Android kernels gaining significant performance improvements. However, current studies as well as implementations of PGO-based Android kernel optimizations still suffer from three problems: 1) optimization inflexibility due to restricted algorithms for generating profiles and for simulating real-world usage scenarios; 2) considerable optimization efforts due to the extensive manual interventions needed; and 3) optimization failures due to kernel version fragmentation.**

This paper presents MEPOF, the *first* modular and end-to-end PGO framework for Android kernels. The MEPOF framework consists of three key components: 1) a tool orchestration, that integrates two novel algorithms for generating profiles, and three methods for simulating real-world scenarios that can be flexibly switched according to the usage scenario; 2) a domain-specific language (DSL) that can specify PGO-based optimization strategies and a corresponding compiler translating the DSL programs into configuration files necessary for optimization; and 3) an adapter that automatically triggers and completes the optimization when the Android kernel version changes.

We have implemented a prototype for MEPOF and have conducted extensive experiments to evaluate its effectiveness, performance, and usability. Experimental results demonstrated that: 1) MEPOF is effective, with performance improvement 9.39% on average; 2) MEPOF is efficient by saving up to 39.07% of time than manual optimizations; and 3) MEPOF is highly usable by requiring only one manual intervention instead of more than 30 manual interventions as in the existing optimization framework.

***Index Terms*—Profile-Guided Optimization, Android Kernel, Optimization Framework**

I. INTRODUCTION

Optimizing the Android kernels is important in improving the performance of the Android system [1], bringing significant advantages to the whole ecosystem, given the wide deployment of Android on billions of diverse devices. Existing optimization frameworks mostly rely on heuristic-based optimization strategies to optimize Android kernels, in which optimization strategies are determined in advance and thus statical, ignoring the runtime feedback an Android kernel might generate during execution. To this end, although traditional heuristic-based optimization methods are effective in performing *static* optimizations, they are often imprecise and rigid due to the lack of detailed runtime information. Worse yet, such a lack of runtime information might even lead to negative optimization results [2].

Recently, Profile-Guided Optimization (PGO) [1], a novel compiler optimization strategy leveraging runtime feedbacks, has been proposed to optimize Android kernels and has shown promising potentials. Specifically, optimizing Android kernels with PGO consists of three key steps: 1) instrumentation; 2) profile generation; and 3) recompilation. First, the target program (Android kernels in this scenario) being optimized is first instrumented with code recording runtime information of the target program (e.g., the frequency a function is invoked). Second, the instrumented target program executes and generates profiles which are collected. Finally, the target program being optimized is recompiled by leveraging the generated profiles [3], to produce an optimized binary.

PGO might bring considerable performance improvements to programs optimized with it, due to its capability of leveraging runtime information. Specifically, prior studies have demonstrated the performance gainings are over 9% for Linux kernels and are up to 20% for multithreaded programs [1] in Android.

Challenges. Unfortunately, although prior studies, as well as engineering efforts, have demonstrated the promising potentials of PGO for Android kernel optimizations, the state-of-the-art PGO frameworks still suffer from three problems: 1) optimization inflexibility due to restricted algorithms for generating profiles and for simulating real-world usage scenarios; 2) considerable optimization efforts due to the extensive manual interventions needed; and 3) optimization failures due to Android kernel version fragmentation. First, PGO frameworks in prior studies lack flexibility. In particular, they are only applicable to specific usage scenarios, due to their integrated and thus rigid algorithms for generating profiles and limited techniques for simulating real-world scenarios. Therefore, when the usage scenario changes, it is difficult for the existing PGO frameworks to change algorithms for profile generation or methods to match the usage scenario.

Second, it is time-consuming, error-prone, and labor-intensive to optimize the Android kernels with the existing PGO frameworks for three reasons: 1) implementing PGO manually is time-consuming, as the complete PGO process is complicated [1], consisting of several distinct phases, including instrumentation, kernel burning, profile collection, and re-

* Corresponding authors.

compilation, which pose distinct implementation requirements; 2) implementing PGO is error-prone due to the complexity in its engineering steps. For example, instrumentation, one of the most complex phases in PGO, includes more than 30 modifications to the Android kernel; even one modification bug will lead to incorrect PGO optimization results; and 3) implementing PGO is labor-intensive. For example, the profile collection phase of PGO requires extensive manual interventions to mimic real-world usage scenarios such as clicking or touching the phone.

Third, the fragmentation, diversity, and frequent kernel updating nature of Android kernels invalids PGO optimizations results. Specifically, optimization strategy built on one version of the kernel cannot be applied to other kernel versions, thereby increasing the implementation efforts.

To address the aforementioned challenges, we propose that an ideal PGO framework for Android kernels should meet the following three requirements:

- (R1) **Flexibility and scalability.** The framework should be flexible in selecting appropriate algorithms for generating profiles and techniques for simulating real-world usage scenarios. Furthermore, the framework should be scalable to potential algorithms and techniques.
- (R2) **End-to-end.** The framework should be end-to-end in generating the optimized kernel image directly from the kernel source code and optimization requirements provided by the end user, minimizing the manual interventions required.
- (R3) **Self-adaptiveness.** The framework should automatically trigger and complete new optimizations when the kernel source code changes, thus avoiding optimization failures.

Our work. In this paper, we proposed MEPOF, the *first* modular and end-to-end PGO framework for Android kernels. To fulfill the above three requirements, MEPOF introduced three key components: 1) a tool orchestration; 2) a novel domain-specific language (DSL) we designed and its corresponding compiler; and 3) an adapter. First, we designed a tool orchestration to integrate diverse algorithms for profile generation and techniques for simulating real-world usage scenarios. On the one hand, to demonstrate the flexibility of the orchestration, it integrated two state-of-the-art algorithms, PGO and Context-Sensitive PGO (CSPGO) [4], and click strategies such as random clicks and traversal-based clicks. On the other hand, to make MEPOF more scalable, we reserved special APIs in tool orchestration to integrate potential algorithms and simulating techniques.

Second, we designed a domain-specific language dubbed G4 and its corresponding compiler g4cc. The language is used to describe the strategies for the entire optimization process. Specifically, the g4cc compiler takes as input the optimization descriptions written in G4 and outputs the required configuration files for the whole optimization process. Next, MEPOF automatically optimizes the Android kernel according to the configuration files and generates an optimized kernel image. Using domain-specific language, only one input (G4 code) is required to obtain the output (optimized Android kernels) without additional manual interventions.

Third, we designed an adapter to adapt different versions of the kernel automatically. First, the adapter selects the desired kernel version based on the configuration. Next, the adapter also keeps track of kernel versions that are still being maintained and automatically download kernel source code and trigger an automated optimization process when a new version is released.

We have implemented a software prototype for MEPOF and have conducted extensive experiments to evaluate its performance, effectiveness, and usability. First, we evaluate the effectiveness of MEPOF generated kernels. And the results demonstrated MEPOF is competitive with manual optimized Android kernels by having 2% difference on average. Second, we conducted four experiments to demonstrate the efficiency of MEPOF by saving up to 39.07% development time compared with manual methods. Third, experimental results demonstrated that MEPOF is usable, by requiring only one human, while manual optimization requires more than 30 interventions.

Contributions. To summarize, this work represents the first step towards designing and implementing a modular and end-to-end PGO framework for Android kernels. The main contributions of our work are as follows:

- **A PGO framework for Android kernel optimizations.** We presented MEPOF, the first end-to-end optimization framework for optimizing Android kernels using PGO. MEPOF meets three requirements : flexibility and scalability, end-to-end, and self-adaptiveness.
- **A prototype implementation of MEPOF.** We implemented a prototype of MEPOF consisting of a novel domain-specific language and its compiler, a tool orchestration, and an adapter.
- **Extensive evaluation of MEPOF.** We conducted extensive experiments to evaluate the effectiveness, efficiency, and usability of MEPOF, demonstrating its practical usefulness.

Outline. The rest of this paper is organized as follows. Section II presents the background of PGO and the challenges in applying it to Android kernels. Section III and IV discuss the overall design and prototype implementation of MEPOF, respectively. Section V presents the evaluations we conducted. Section VI discusses the limitation of this work and directions for future work. Section VII describes related work, and Section VIII concludes.

II. Background and Challenges

To be self-contained, we present, in this section, necessary background information on PGO and PGO-based Android kernel optimizations (Section II-A), and the challenges of applying PGO to Android kernel optimizations (Section II-B).

A. Profile-Guided Optimization

Profile-Guided Optimization (PGO), also called feedback-directed optimization (FDO) [5], is a compilation strategy

that effectively improves program performance [6] [7] by leveraging runtime information called *profiles*.

PGO workflow. The workflow of PGO consists of three typical phases: 1) instrumentation; 2) profile generation; and 3) program recompilation. First, in the instrumentation phase, the compiler instruments the target program being optimized with instrumentation code. The instrumentation code is highly dependent on optimization goals. For example, to perform loop optimizations [8], the compiler inserts execution counting code at each branch point in the target program being optimized, to count and record the execution frequencies of each branch when the target program executes. The program being optimized as well as the instrumented code will then be compiled into instrumented binaries. Second, in the profile generation phase, the instrumented binaries are executed to generate runtime information called profiles, containing important runtime information such as the execution frequency of a function or a loop. Third, in the program recompilation phase, the compiler will optimize the target program for a second time by leveraging the profile generated. For example, the compiler will inline a function whose call frequency exceeds some specified threshold, thereby improving program performance.

PGO history. PGO has been well studied with a long history due to its potential to leverage runtime profiles in guiding optimizations. Studies of PGO date back at least to the 1960s [9] [10] [11]. Recently, with better hardware support [3], PGO has been extensively studied [12] [13] and has been successfully used in optimizing a large spectrum of real-world systems such as Chrome [14], PHP [15], .NET [16], Firefox [17], Service Mesh, and even Linux kernels [18]. The application of PGO to optimizing these systems brings considerable performance improvements. For example, in optimizing .NET Core 2.0, PGO brings a performance increase up to 21.33% [19].

Compiler support. As a promising compiler optimization, PGO has been well supported by mainstream compilers. For example, recent releases of GCC [20] and LLVM [21] have complete support of PGO and are still developing novel features. Compared with traditional optimizations, compiler support of PGO optimization brings two grand advantages by: 1) making optimization more accurate; and 2) revealing new optimization opportunities. First, PGO technology is capable to obtain runtime profiles, which are absent in static optimization technology. and the existing optimization options can use this information to improve their optimization effect. For example, the branch frequency and other information contained in the profile file can improve the probability of branch prediction. Second, the runtime information in the generated profiles has the potential to reveal new optimization opportunities. For example, by leveraging function execution frequency information in the generated profiles, the compiler can determine which functions are frequently called, thus might trigger inline optimization to reduce the number of function calls leading to better performance.

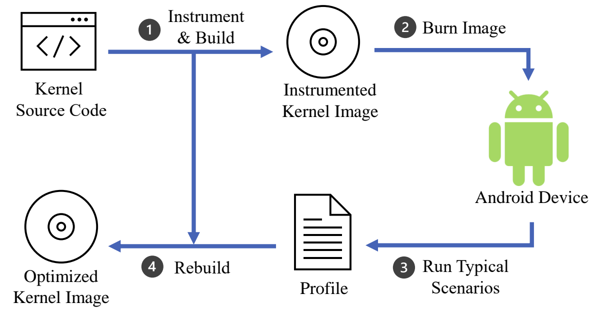


Fig. 1: The PGO workflow for optimizing Android kernel.

B. Challenges for PGO-based Android Kernel Optimization

While, PGO can be used to optimize Android kernels in principle, this optimization process faces several challenges due to the unique characteristics of Android, as Fig. 1 demonstrated. First, Android kernel source code is instrumented and compiled to generate an instrumented kernel image (❶), which is burned into a target device such as a physical phone or a simulator (❷). It challenging to finish this step as a considerable amount of modifications to the source code are needed, which is not only laborious but also error-prone.

Second, typical scenarios are executed to generate and collect profiles (❸). This phase is challenging as a significant amount of mobile applications need to be executed under diverse scenarios to simulate real-world usages.

Third, profiles are leveraged for Android kernel source recompilation (❹), to generate an optimized kernel image. In this phase, it is challenging to select most appropriate optimization options to obtain optimized kernel images with maximum possible performance improvement.

Finally, the generated profiles become invalid when the Android kernel source code undergoes any changes, as profiles stored runtime information such as the execution frequencies of a function. Hence, when the Android source code changes, the runtime information in the already collected profiles might not be applicable to subsequent optimizations. In the meanwhile, Android sources are updated frequently, making it challenging for PGO to account for this updating frequency.

III. Design

In this section, we present the design of the MEPOF. We first discuss the architecture of MEPOF (Section III-A), then present each component including the G4 domain-specific language (III-B), the g4cc compiler (III-C), tool orchestration (III-D), adapter (III-E), and evaluation (III-F).

A. Architecture

The overall architecture of MEPOF is presented in Fig. 2, which consists of four key components: 1) the G4 language and its compiler g4cc; 2) the adapter; 3) the tool orchestration; and 4) the measurement. First, MEPOF takes as input both the Android kernel sources and a G4 program describing optimization strategies. The Android kernel sources are either

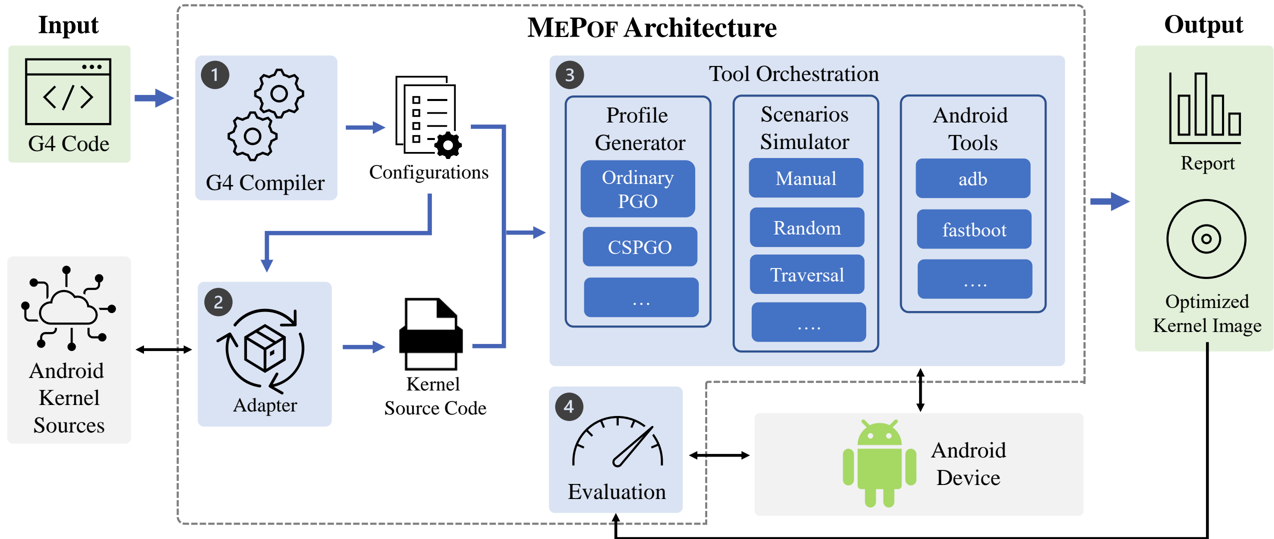


Fig. 2: The Architecture of MEPOF.

the official or vendor-supplied releases, without any special configurations or modifications. This design decision not only makes it easier for end users to use MEPOF, but also makes MEPOF more general and scalable to process any version of Android kernels without intrinsic difficulty. The G4 program, developed according to the syntax of the G4 domain-specific language we designed, describes optimization strategies to be used in PGO optimization and will be discussed in detail next (Section III-B).

Second, the g4cc compiler (❶) we designed takes as input a G4 program and compiles it to configuration files to be used in the subsequent PGO optimizations.

Third, the adapter module (❷) takes as input the generated configuration files and downloads or fetches the specified version of the Android kernel for subsequent processing, in a fully automated manner.

Fourth, the tool orchestration module (❸) takes as input both the Android kernel source code and the automatically generated configuration files, orchestrates specific PGO algorithms, simulation strategies, and Android tools required to trigger the required PGO optimizations and generate an optimized Android kernel image automatically.

Finally, the evaluation module (❹) takes as input the optimized Android image and evaluates this image according to the measurement metrics from the configuration file. As output, this module will generate a measurement report to evaluate whether the optimization has met the goal.

B. The G4 *Domain-specific Language*

In this section, we present G4, a domain-specific language to specify PGO-based optimization strategies for Android kernels. To address the aforementioned challenges of PGO-based Android kernel optimizations (Section II-B), we have three design goals for the G4 domain-specific language: 1) flexibility; 2) end-to-end; and 3) self-adaptiveness. First, the G4 language should be flexible in specifying possible PGO optimization strategies. For example, G4 should easily express diverse optimization algorithms, such as standard PGO or CSPGO, so that algorithm switching has zero cost. Second, the G4 language should express end-to-end decisions to reduce manual interventions. To this end, the G4 language should cover every stage of the PGO optimization process.

$$
\begin{array}{llcl}
\text{Algorithm} & l & ::= & \texttt{pgo} \mid \texttt{cspgo} \mid \ldots \\
\text{Click} & k & ::= & \texttt{manual} \mid \texttt{rand} \mid \texttt{trav} \mid \ldots \\
\text{Evaluation} & a & ::= & \texttt{eval}\ img\ \overrightarrow{metric} \\
\text{Recompile} & c & ::= & \texttt{recomp}\ path\ p\ \overrightarrow{op} \\
\text{Adapter} & y & ::= & \texttt{adap}\ url\ l\ k\ \overrightarrow{ap}\ \overrightarrow{op} \\
\text{Merge} & m & ::= & \texttt{merge}\ \overrightarrow{p}\ \overrightarrow{w} \\
\text{Profile} & r & ::= & \texttt{profile}\ path\ l\ k\ \overrightarrow{ap} \\
\text{Optimization} & o & ::= & \texttt{opt}\ path\ l\ k\ \overrightarrow{ap}\ \overrightarrow{op} \\
\text{Statement} & s & ::= & o \mid r \mid m \mid y \mid c \mid a \\
\text{Program} & g & ::= & \overrightarrow{s}
\end{array}
$$

Fig. 3: Syntax of G4.

With these design goals, in Fig. 3, we present the syntax of G4 using a context-free grammar. A G4 program g consists of a list of statements s, where the notation $\overrightarrow{s}$ stands for zero or more of statements s (i.e., a Kleene closure). According to the specific operation it performs, a statement s can be classified into six categories: an optimization o, a profile r, a merge m, an adapter y, a recompile c, or an evaluation a.

All categories of statements s have similar structures by starting with an operation followed by zero or more operands. In particular, an optimization statement o consists of an oper-

Algorithm 1 : Compilation algorithm for G4.

Input: p: a G4 program.
Output: c: the optimization configurations.
1: **procedure** COMPILE(p)
2: ast = parse(p)
3: c = generate(ast)
4: **return** c

ation flag opt and five parameters: the string parameter $path$ stands for the location of the Android kernel source code. The parameter l stands for the profile generation algorithm, and parameter k represents the method for collecting the profiles. The parameter $\overrightarrow{ap}$ is a list of application ap that are used for simulating real-world scenarios. The parameter $\overrightarrow{op}$ is a list of optimization option op used in the recompilation phase.

A profile statement r is used to generate the configurations required for program analysis, consisting of an operator `profile` followed by four parameters. A merge statement m is used to generate the configurations required for merging profiles and consists of an operator `merge` followed by two parameters: $\overrightarrow{p}$ and $\overrightarrow{w}$. The parameter $\overrightarrow{p}$ is a list of profile p, and $\overrightarrow{w}$ is a corresponding list of proportions used for merging profiles.

An adaptor statement y is used to generate the configurations for optimizing the specified kernel automatically, consisting of an operator `adap` followed by five parameters. Among them, the string parameter url stands for the address where the specified Android kernel resides. A recompile statement r can be used to generate the configurations required for recompilation and consists of an operator `recomp` followed by three parameters. Among them, the parameter p is the profiles guiding recompilation optimization. An evaluation statement a is used to generate the configurations for evaluating the kernel performance and consists of an operator `eval` with two parameters, among which the first parameter img denotes the location of the kernel image, whereas the second parameter $\overrightarrow{metrics}$ stands for a list of metrics against which the evaluation should be performed.

C. The g4cc Compiler

The g4cc compiler takes as input a G4 program and compiles it to optimization configuration files. The compiler g4cc follows a modular design, consisting of three key phases, as Algorithm 1 presents: 1) the front-end; 2) the abstract syntax trees; and 3) the code generator.

First, the front-end of g4cc reads in the source code of a G4 program and constructs an abstract syntax tree for subsequent processing. As the syntax of G4 is relatively straightforward, we have decided to make use of a handwritten scanner and parser to parse the G4 sources. The scanner makes use of a transition graph algorithm and the parser uses a recursive decedent algorithm which are both standard compiler algorithms and thus deserve no further discussion. Another possible design choice is to leverage automatic generators (e.g., lex [22] and bison [23]) to build the front-end. While this choice is promising in saving some manual effort, it is arguably of no dramatic difference given the simplicity of the G4 language.

Second, the g4cc compiler constructs an abstract syntax tree for a G4 program, which is the main internal data structure for subsequent processing. We also impose certain restrictions on user input to simplify source processing. For example, for the input $\overrightarrow{op}$, the user should place a space symbol ";" between two optimization options, and a symbol "." at end to terminate the corresponding statement.

Third, the g4cc compiler generates optimization configuration files as output, from the constructed abstract syntax tree, by a postorder tree traversal.

D. Tool Orchestration

As shown in Fig. 2, the tool orchestration module takes as input both the Android kernel and the configurations to generate a profile for subsequent kernel optimizations. More precisely, the orchestration module consists of three submodules: 1) profiling algorithm selection; 2) scenario simulations; and 3) Android tool integration. The details of the design are presented in Algorithm 2.

First, the profiling algorithm selection module selects the corresponding PGO algorithm according to the configurations, and then integrates these algorithms into optimizations. For example, if we want to use the CSPGO algorithm to obtain the profile, we only need to set the statement parameter to `cspgo` in the G4 code. To enhance the flexibility of the framework, we have designed APIs to integrate algorithms. Hence, the optimization framework only needs to call the corresponding API without extensive modification of the framework source code, to incorporate new algorithms.

Second, the Android kernel k is instrumented according to the strategy specified by the profile p, to obtain a new kernel k'. In this step, the kernel compilation script needs to be modified to add support for instrumentation. It is worthy noting that performing global instrumentation on the kernel source code may cause a kernel crash. Therefore, it is necessary to exclude modules that may lead to such crashes. Next, the instrumented kernel k' is compiled by a PGO-enabled compiler to obtain a PGO-enabled kernel image i.

Third, the instrumented kernel image i is executed on Android devices, to collect runtime profiles r, according to the simulations s and Android tool configuration t. The Android devices not only include physical devices such as mobile phones or tablets, but also virtual simulators such as Bluestacks [24] or Virtualbox [25]. To generate and collect runtime profiles, the instrumented Android kernel image i is first flushed to the target Android devices via specific tools such as adb [26] or fastboot [27]. Then, the target device is rebooted and benchmarks are executed on this instrumented kernel i, with the simulation strategy s and the selected toolkit t. MEPOF is flexible in incorporating diverse strategies for simulating real-world scenarios. In particular, MEPOF supports flexible clicking strategies such as `manual`, `rand`, and `trav`, which are indispensable in executing Android benchmarks. For

Algorithm 2 : Orchestration Algorithm

Input: c: the configuration; k: the kernel source
Output: i': optimized kernel image

```
1:  procedure ORCHESTRATION(c, k)
2:      (p, s, t) = analyze(c)
3:      k' = instrument(k, p)
4:      i = compile(k')
5:      r = collectProfile(i, s, t)
6:      i' = compileWithProfile(k, r)
7:      return i'
8:  procedure COLLECTPROFILE(i, s, t)
9:      flashImage(i, t)
10:     reboot()
11:     p = runScenarios(i, s, t)
12:     return p
```

example, the click strategy based on traversal will click each button in the target APP in a specified order. This technique can guarantee consistency of the click sequence during a simulation, covering as many execution paths as possible.

E. Adapter

The adapter is utilized to adapt to the fragmentation of Android kernels, thus achieving the self-adaptiveness goal. First, to address the aforementioned optimization incompatibility issues caused by Android kernel fragmentation and diversity, we have built in multiple versions of kernel source code into the adapter. During optimization, the adaptor module automatically matches and selects the corresponding version based on the generated configurations. Second, to mitigate for optimization failures caused by kernel updates, the adapter tracks the kernel release website and automatically downloads the latest source code when a new version is released, then triggers an automated optimization process.

F. Evaluation

The evaluation module of MEPOF takes as input the PGO-optimized Android kernel and a set of benchmarks, and outputs measurement results by executing these benchmarks.

The evaluation module automatically evaluates the performance of the instrumented Android kernel against specific evaluation metrics and benchmarks. Although some benchmarks (e.g., Antutu [28] or Geekbench [29]) for evaluating the performance of Android phones are available, they are CPU/GPU intensive, and are mainly used to test the performance of the hardware. To this end, to evaluate the performance of the Android kernel, we follow existing studies on evaluating Android kernel performance [1] and select five performance metrics: system call, system call overhead [30], process creation speed, pipe throughput, and file copy speed. First, as bridges between the user space and the kernel space, the system calls are used to evaluate Android kernels. For example, to measure process creations, we evaluate the `execl()` system call. Second, the system call overhead affects the performance of the kernel. Thus we record the duration between a specific system call enters and leaves the kernel space, to evaluate the overhead of the system call. For example, we evaluate the `getpid()` system call to measure its execution time. Third, the Android kernel is large and complex software consisting of many components including process management, interprocess communication (IPC), and file systems, etc. Therefore, we use process creation speed to measure process performance and pipe throughput to measure interprocess communication performance. Furthermore, file operations, including the creation, file copying, reading and writing of files, are important and ubiquitous, so we use file copy speed to measure the performance of file systems.

To use the evaluation module to evaluate the optimized Android kernel, the evaluation statement a in the G4 language is used. The first parameter img of the statement a specifies the kernel image, and the second parameter $\overrightarrow{metrics}$ represents the performance metrics used in this evaluation. For example, in the sample statement `eval /sys/out/opt.img file;pipe.`, MEPOF will evaluate the kernel image `/sys/out/opt.img`, in terms of the file system (`file`) and pipe (`pipe`) performance.

G. Android Image and Report Generation

After finishing all the above phases, MEPOF generates as outputs the PGO-optimized Android kernel image, as well as the corresponding final evaluation report for subsequent analysis.

IV. IMPLEMENTATION

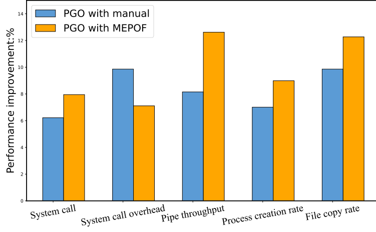
We have implemented a software prototype for MEPOF. We have leveraged a handwritten strategy to implement the compiler g4cc, including its scanner, parser, abstract syntax trees, and code generator. We used the `diff` tool [31] to generate the patches for PGO and CSPGO instrumentations. We implemented the orchestration module using a Python script, to drive the whole optimization process. We leveraged the Monkey tool [32] for random clicking and the Droidbot [33] tool for traversal clicking, to simulate real-world usage scenarios. We used Android platform tools including `adb` and `fastboot`, to transfer files and burn the kernel image to experimental devices. We wrote a crawler in Python to monitor the kernel version releases. For Android kernel updating, the framework will use the crawler to automatically download the kernel source code and change to trigger the automatic optimization process.

V. EVALUATION

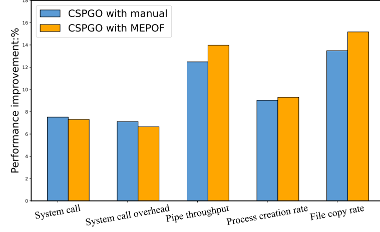
In this section, we present experiments to evaluate MEPOF. We first present the research questions guiding the evaluation (V-A). Next, we evaluated MEPOF in terms of effectiveness, efficiency, and usability, respectively (Section V-D to V-F).

A. Research Questions

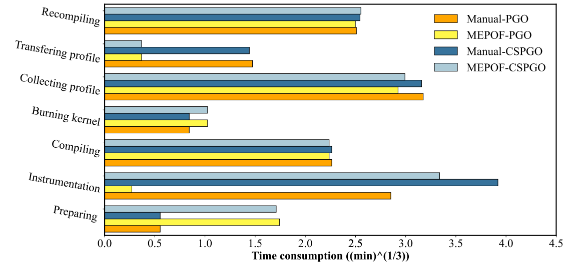
By presenting the experimental results, we mainly investigate the following research questions:



(a) The effectiveness of MEPOF using PGO.



(b) The effectiveness of MEPOF using CSPGO.



(c) The performance of MEPOF.

Fig. 4: The effectiveness of MEPOF PGO, CSPGO, and performance of MEPOF, respectively.

RQ1: Effectiveness. Is MEPOF effective in producing optimized kernels with significant performance improvements?

RQ2: Efficiency. To what extent can MEPOF improve the efficiency in optimize Android kernels?

RQ3: Usability. Is MEPOF usable to reduce the manual efforts and interventions in the entire optimization process?

B. Experimental Setup

All the experiments and measurements are performed on a server with an Intel Core i7-12700k processor and 64 GB of RAM. The operating system of the server is Ubuntu 20.04, with Clang version 11.0.1. The Android kernel version is msm-5.4, which is in turn based on Linux kernel 5.4. In addition, we use Oneplus 9 Pro 5G as a physical phone for evaluation.

It is necessary to set affinity for the Android phones and fix the CPU frequency, and to turn off power-saving mode to accurately measure the performance of the kernel. It is also necessary to turn on the airplane mode and restart and clear the phones before each test to control the experimental variables.

C. Benchmarks

The are no publicly available benchmarks for evaluating PGO-optimized Android kernels to the best of our knowledge. We thus took the first step to manually construct a micro-benchmark, including a total of 8 test cases. These test cases are classified into five categories, which are used to test the five performance metrics as discussed above (Section III-F): 1) to evaluate system call performance, we created test cases to count the execution frequencies of system call per second; 2) to evaluate system call overhead, we record the duration that a system call takes between entering and leaving the kernel space; 3) to evaluate pipe performance, we created test cases that transmitting 128/512 bits data using pipes and record the number of transmission within one second, respectively; 4) to evaluate process creation performance, we created test cases counting the number of process creation and destroy per second; and 5) to evaluate file operation performance, we created test cases to count the number of characters read from a file, where the file size and buffer size combinations are 512B+128B, 1024B+256B, and 1K+8K, respectively.

D. RQ1: Effectiveness

To answer **RQ1** by investigating the effectiveness of MEPOF, we used manual methods and MEPOF to optimize the Android kernel, respectively. Then, we burned these optimized kernels into the same experimental Android device to measure their performance. The experiment was carried out 5 rounds, to calculate the average results.

First, we performed normal PGO to optimize the Android kernel using manual and MEPOF, respectively, and then performed performance evaluations on the two optimized kernels. The experimental results are shown in Fig. 4a. Taking the unoptimized kernel performance as the baseline, the y-axis in the figure represents the percentage of performance improvements. The results show that manually optimized Android kernel has lower overhead in terms of system calls, but triggers higher overhead in terms of the other four metrics. Among them, for the pipe performance, the improvement of MEPOF is 4.47% higher than the manual method. On an average, the manual method has an improvement by 8.02%, whereas MEPOF has an improvement by 9.39%. Hence, MEPOF outperforms the manual method, although the difference is insignificant.

Second, we utilized CSPGO strategy to optimize the Android kernels using manual and MEPOF, respectively, and performed performance experiments on the two optimized kernels. The experimental results are shown in Fig. 4b. On an average, the manual method has an performance improvement by 9.13%, whereas MEPOF improved by 10.1%. The performance improvements are close.

In summary, the performance improvement of using MEPOF is in par with the manual optimization, therefore demonstrating the effectiveness of MEPOF.

E. RQ2: Efficiency

To answer **RQ2** by demonstrating the efficiency of MEPOF, we explore to what extent MEPOF can save development and experiment time than the manual optimization strategy for Android kernel optimizations.

Specifically, in this experiment, we used the following four combinations of optimization strategies: manual PGO, manual CSPGO, PGO with MEPOF, and CSPGO with MEPOF. For each optimization strategy, we optimized the Android kernel then recorded the time spent on each stage of the optimization

TABLE I: Manual interventions in manual methods and MEPOF, respectively.

Phases	Manual	MEPOF	Interventions
Preparation	•	•	/
Instrumentation	•	◦	1,241 lines of code
Compilation	•	◦	3 instructions
Burning kernel	•	◦	2 instructions
Collecting profile	•	◦	n clicks
Transferring profile	•	◦	2 instructions
Recompilation	•	◦	3 instructions

process. In order to measure the time consumption for each phase more accurately, we divide the complete process of PGO for Android kernel optimization into the following 7 specific stages: 1) optimization preparation; 2) instrumentation; 3) instrumented Android kernel compilation; 4) instrumented Android kernel image burning; 5) profile collection by executing usage scenarios; 6) profile transfer and conversion; and 7) recompiling Android kernel source using profiles. The experiment was repeated 5 rounds, to calculate the average time for each phase. The experimental results are shown in Fig. 4c.

The results show that the MEPOF has significantly improved the development and experiment performance for the two phases: instrumentation and profile transfer. When using CSPGO to optimize the Android kernel, the instrumentation phase takes 60.2 minutes for manual method to finish, whereas it takes 37.3 minutes for MEPOF to finish, saving 22.9 minutes. Overall, MEPOF can improve the efficiency up to 39.07% compared with manual PGO; and up to 20.69% compared with manual CSPGO. To summarize, it is more efficient to use MEPOF to optimize the Android kernel than the manual method.

F. RQ3: Usability

To answer **RQ3** by demonstrating the usability of MEPOF, we counted and compared the manual interventions between the manual PGO method and MEPOF. In this experiment, manual interventions include not only the instructions input, but also any forms of user efforts in conducting this experiment such as source code modifications, touching screens, and/or clicking buttons.

To gain a thorough understanding of manual interventions in each phase, we measure manual interventions in each phase separately with distinct metrics, as shown in in TABLE I, where the symbol • indicates that the corresponding phase requires manual intervention, whereas the symbol ◦ indicates that no manual interventions are needed.

The experimental results show that the preparation phase of both manual method and MEPOF need manual interventions. Except for the preparation phase, MEPOF does not need any manual interventions but the manual method needs manual interventions in each of its phases. Specifically, in the manual method, a manual modification of 1,241 lines of source code is needed. And each other phases, including compilation, burning the kernel, transferring profiles, recompilation, need 2 or 3 manual instructions, respectively. On the contrary, MEPOF does not need any interventions during these phases. These experimental results demonstrated that MEPOF is practically usable in optimizing Android kernels.

VI. DISCUSSION

In this section, we discuss some limitations of MEPOF and possible directions for future work. It should be noted that this work represents the first step towards designing and implementing a practical PGO-based optimization framework for Android kernels.

Profile generations. PGO-based optimizations rely on the effective generation of profiles, and prior studies have proposed two categories of techniques: 1) instrumentation [1]; and 2) hardware sampling [3]. In this work, we have leveraged the instrumentation-based profile generation technique, and the experimental results demonstrated that this approach is effective. On the other hand, it might be promising to adopt the hardware sampling technique for profile generation, as it is more lightweight than instrumentation. However, hardware sampling is a relatively new hardware technology that is only available on recent ARM64 CPUs with the embedded trace macrocell (ETM) [34] (the different but equivalent technology on x86_64 CPUs is the LBR function [35]), thus, to the best of our knowledge, few platforms have this hardware feature shipped. In future work, we plan to further explore the hardware sampling techniques for profile generation, when we have the necessary proprietary computing resources. However, it should be noted that the hardware sampling technique is orthogonal to instrumentation, as Android kernels can always be sampled independent of they are instrumented or not.

Simulations. Simulating real-world scenarios is important to generate profile with high accuracies on mobile platforms. In this work, we have explored three techniques for simulations, and the experimental results demonstrated they are effective. In addition, recent studies have proposed other simulation techniques (e.g., deep learning-based simulations [36]), which have the potential to generate more accurate profiles. In future work, we plan to investigate the deep learning-based approach for profile generations. Fortunately, the architecture of MEPOF (Fig. 2) is neutral to any specific simulation techniques deployed, thus, it should be of no intrinsic difficulty to integrate new simulation techniques into MEPOF.

Optimization options. Optimization options and their correction combinations are indispensable to generate efficient Android kernels from profiles. In this work, MEPOF provided full support for all widely used compiler optimization options, including static optimization options (e.g., O2, O3, and Os), and link-time options (e.g., FullLTO). The support of a full range of optimizations not only makes PGO optimizations more effective, but also makes it smooth to leverage existing optimization frameworks without any modifications. Meanwhile, recent studies have proposed more promising optimization opportunities. For example, BOLT [37] [38], a novel link-time binary optimization, has been extensively studied and experimental results demonstrated considerable

speedups due to this optimization strategy (up to 50% for Linux binaries [39]). In future work, we plan to integrate BOLT optimizations into MEPOF, which may make PGO optimizations more effective.

Push-button optimizations. To make MEPOF flexible and scalable, we have defined a domain-specific language G4, with which developers can write G4 programs to specify strategies in PGO-based optimizations. The G4 programs are then compiled, by the compiler g4cc, into configurations for optimizations. While G4 is intentionally designed to have a clean syntax and intuitive semantics, it does have a learning curve for users, especially for Android developers who have little or no PGO optimization background knowledge. Thus, in future work, we plan to design syntax sugar on top of G4 to make programming more accessible. Ideally, end developers only need to describe optimization goals in a "push-button" style [40], and the corresponding G4 programs can be synthesized automatically. Furthermore, it is also promising to synthesize g4cc in an automated manner (similar to JitSynth [41]), and we also leave it a future work.

VII. RELATED WORK

In recent years, there have been a significant number of studies on PGO-based optimizations, both for Android kernels and for general software systems. However, the work in this paper represents a novel contribution to this field.

Profile generations. Profile generations have been extensively studied, with two approaches proposed: instrumentation and hardware sampling. Among instrumentation algorithms, Knuth [10] proposed the least counter algorithm in 1973. By limiting the scope of the instrumentation, the extra runtime overhead of the instrumented program is reduced considerably. Traditional PGO instrumentation is context insensitive. To address this limitation, Xu et al. [4] proposed CSPGO, a novel instrumentation to achieve better effects by leveraging contexts. Ellis et al. [42] proposed a lightweight PGO (IRPGO) to reduce runtime overhead, making it more suitable for mobile devices. Among the sampling algorithms, Chen et al. [14] proposed the AutoFDO algorithm, which generates profile by collecting the last branch record (LBR). Diego Novillo et al. [43] proposed SamplePGO, which uses an external sampling analyzer to obtain a profile. The runtime overhead of this algorithm is negligible, as no instrumentation is used.[]

PGO-based optimizations. PGO-based optimizations have been studied extensively. Wang et al. [12] used PGO to study dataflow prediction. Huang et al. [44] used PGO to study indirect branches in a binary translator. Homescu et al. [13] used PGO technology to reduce the performance overhead caused by software diversity in defending against code reuse attacks. Williams et al. [45] used PGO to optimize workloads on x86 platforms. Lee et al. [46] proposed a lightweight instrumentation at machine IR level using LLVM. However, optimizing large software such as the Android kernels with PGO is time-consuming, laborious and error-prone. Therefore, the study in this paper is orthogonal to existing studies by proposing an effective optimization framework.

VIII. CONCLUSION

This paper presents MEPOF, the first modular and end-to-end framework for PGO-based Android kernel optimizations. To make MEPOF flexible, end-to-end, and self-adaptive, we introduced three novel techniques, including a tool orchestration to flexibly switch PGO algorithms and strategies for different usage scenarios, a novel domain-specific language G4 and its compiler to specify optimization strategies, and an adapter to address optimization failures caused by kernel version fragmentation and updates. We have implemented a software prototype for MEPOF and conducted extensive experiments to evaluate it. The experimental results demonstrated that MEPOF is effective, efficient, and highly usable. This work represents a new step towards applying PGO-based optimizations to Android kernels, thus making Android, the most pervasive kernel for today's mobile computing, more efficient and competitive.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable feedback. This work is partially supported by the National Natural Science Foundation of China (No.62072427, No.12227901), the Project of Stable Support for Youth Team in Basic Research Field, CAS (No.YSBR-005), Academic Leaders Cultivation Program, USTC.

REFERENCES

[1] P. Yuan, Y. Guo, X. Chen, and H. Mei, "Device-specific linux kernel optimization for android smartphones," in *2018 6th IEEE International Conference on Mobile Cloud Computing, Services, and Engineering (MobileCloud)*. IEEE, 2018, pp. 65–72.

[2] T. A. Khan, I. Neal, G. Pokam, B. Mozafari, and B. Kasikci, "Dmon: Efficient detection and correction of data locality problems using selective profiling." in *OSDI*, 2021, pp. 163–181.

[3] B. Wicht, R. A. Vitillo, D. Chen, and D. Levinthal, "Hardware counted profile-guided optimization," *arXiv preprint arXiv:1411.6361*, 2014.

[4] "Cspgo," https://reviews.llvm.org/D54175.

[5] 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*, 2010, pp. 42–52.

[6] "Ibm's pgo documentations," https://www.ibm.com/ docs/ en/ openxl-fortran-aix/ 17.1.0?topic=compatibility-profile-guided-optimization-pgo,.

[7] "Intel's pgo documentation," https://www.intel.com/ content/www/us/en/develop/ documentation/cpp-compiler-developer-guide-and-reference/ top/optimization-and-programming/profile-guided-optimization-pgo.html.

[8] A. Aslam and L. Hendren, "Mcflat: a profile-based framework for matlab loop analysis and transformations," in *Languages and Compilers for Parallel Computing: 23rd International Workshop, LCPC 2010, Houston, TX,*

USA, October 7-9, 2010. Revised Selected Papers 23. Springer, 2011, pp. 1–15.

[9] C. Apple, "Evaluation and performance of computers: the program monitora device for program performance measurement," in *Proceedings of the 1965 20th national conference*, 1965, pp. 66–75.

[10] D. E. Knuth, "An empirical study of fortran programs," *Software: Practice and experience*, vol. 1, no. 2, pp. 105–133, 1971.

[11] D. E. Knuth and F. R. Stevenson, "Optimal measurement points for program frequency counts," *BIT Numerical Mathematics*, vol. 13, no. 3, pp. 313–322, 1973.

[12] L. Wang, H. An, Y. Ren, and Y. Wang, "Profile guided optimization for dataflow predication," in *2008 13th Asia-Pacific Computer Systems Architecture Conference*. IEEE, 2008, pp. 1–8.

[13] A. Homescu, S. Neisius, P. Larsen, S. Brunthaler, and M. Franz, "Profile-guided automated software diversity," in *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2013, pp. 1–11.

[14] D. Chen, D. X. Li, and T. Moseley, "Autofdo: Automatic feedback-directed optimization for warehouse-scale applications," in *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, 2016, pp. 12–23.

[15] "Pgo on php," https://devblogs.microsoft.com/cppblog/speed-up-windows-php-performance-using-profile-guided-optimization-pgo/.

[16] "Pgo on .net," https://devblogs.microsoft.com/ dotnet/ conversation-about-pgo/.

[17] "Pgo on firefox," http://swiftweasel.tuxfamily.org/.

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

[19] "Pgo on .net2.0," https://devblogs.microsoft.com/ dotnet/ profile-guided-optimization-in-net-core-2-0/.

[20] "Gcc support for pgo," https://gcc.gnu.org/ onlinedocs/ gcc/ Optimize-Options.html.

[21] "Llvm support for pgo," https://clang.llvm.org/ docs/ UsersManual.html #profile-guided-optimization.

[22] "Lex," https://www.visionaid.co.uk/lex-software.

[23] "Bison," https://www.gnu.org/software/bison/.

[24] "Bluestacks," https://www.bluestacks.com/.

[25] "Virtualbox," https://www.virtualbox.org/.

[26] "Android debug bridge (adb)," https:// developer. android. com/studio/ command-line/ adb.

[27] "Fastboot," https://source.android.com/ docs/ setup/ build/ running.

[28] "Antutu," https://www.antutu.com/.

[29] "Geekbench," https://www.geekbench.com/.

[30] L. Gerhorst, "Flexible and low-overhead system-call aggregation using bpf," 2021.

[31] "Diff," https://git-scm.com/docs/git-diff.

[32] "Monkey," https://developer.android.com/studio/test/other-testing-tools/monkey.

[33] "Droidbot," https://github.com/honeynet/droidbot.

[34] "Pgo on arm64 cpu." https://developer.arm.com/ documentation/ 101458/ 2210/ Optimize/ Profile-Guided-Optimization–PGO-?lang=en.

[35] "Lbr," https://lwn.net/Articles/680985/.

[36] Y. Li, Z. Yang, Y. Guo, and X. Chen, "Humanoid: A deep learning-based approach to automated black-box android app testing," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 1070–1073.

[37] M. Panchenko, R. Auler, B. Nell, and G. Ottoni, "Bolt: a practical binary optimizer for data centers and beyond," in *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2019, pp. 2–14.

[38] "Bolt," https://github.com/ llvm/ llvm-project/ tree/ main/ bolt.

[39] "Bolt optimization effect," https://www.phoronix.com/ news/ LLVM-Lands-BOLT.

[40] L. Nelson, H. Sigurbjarnarson, K. Zhang, D. Johnson, J. Bornholt, E. Torlak, and X. Wang, "Hyperkernel: Push-button verification of an os kernel," in *Proceedings of the 26th Symposium on Operating Systems Principles*, 2017, pp. 252–269.

[41] J. Van Geffen, L. Nelson, I. Dillig, X. Wang, and E. Torlak, "Synthesizing jit compilers for in-kernel dsls," in *Computer Aided Verification: 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21–24, 2020, Proceedings, Part II*. Springer, 2020, pp. 564–586.

[42] "Irpgo," https://discourse.llvm.org/ t/instrprofiling-lightweight -instrumentation/ 59113.

[43] D. Novillo, "Samplepgo-the power of profile guided optimizations without the usability burden," in *2014 LLVM Compiler Infrastructure in HPC*. IEEE, 2014, pp. 22–28.

[44] J.-S. Huang, W. Yang, and Y.-P. You, "Profile-guided optimisation for indirect branches in a binary translator," *Connection Science*, vol. 34, no. 1, pp. 749–765, 2022.

[45] D. Williams-King and J. Yang, "Codemason: Binary-level profile-guided optimization," in *Proceedings of the 3rd ACM Workshop on Forming an Ecosystem Around Software Transformation*, ser. FEAST'19. New York, NY, USA: Association for Computing Machinery, 2019, pp. 47–53. [Online]. Available: https://doi.org/10.1145/3338502.3359763

[46] K. Lee, E. Hoag, and N. Tillmann, "Efficient profile-guided size optimization for native mobile applications," in *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction*, ser. CC 2022. New York, NY, USA: Association for Computing Machinery, 2022, pp. 243–253. [Online]. Available: https://doi.org/10.1145/3497776.3517764