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

Keyuan Zong Baojian Hua* Yang Wang* Shuang Hu Zhizhong Pan
School of Software Engineering
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 profile and simulating real-world scenarios; 2) considerable optimization efforts due to the extensive manual interventions needed; and 3) optimization may fail due to kernel versions are fragmented and iterative.**

**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 profile, 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 optimized Android kernel performance averaging a 9.39% improvement; 2) MEPOF is efficient by saving up to 39.07% of time compared to manual optimization; 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 has a positive impact on improving the performance of the Android system [1], thus bringing significant advantages to the whole ecosystem, given the wide adoptions of the Android systems on billions of diverse devises. Traditionally, heuristic-based optimizations are employed to optimize Android kernels, in which optimization strategies are determined statically, ignoring the runtime feedbacks an Android kernel generates during execution. To this end, although traditional heuristic-based optimization methods are effective in performing static optimizations, they are often imprecise due to the lack of detailed runtime information. Worse yet, such imprecision during optimization might lead to negative optimizations in some cases [2].

Recently, Profile-Guided Optimization (PGO) [1], a novel compiler optimization technique leveraging runtime feedback, has been proposed to optimize Android kernels and has shown promising potential. Technically, in utilizing PGO, the code for collecting profile is first inserted into the target program to be optimized, and then the profile is collected while running the target program after instrumenting, and finally the target program is recompiled and optimized using the profile[3].

Due to PGO's capability of leveraging runtime feedback, it often brings considerable performance improvements to programs optimized with PGO enabled. In particular, prior studies have demonstrated the performance improvements are over 9% for Linux kernels in Android smartphones and up to 20% for multithread programs[1].

**Problems.** Unfortunately, although prior studies, as well as industry efforts, have demonstrated the promising potential of PGO for Android kernel optimizations, the state-of-the-art PGO frameworks still suffer from three problems: 1) the process lacks flexibility; 2) the process is time-consuming, error-prone, and labor-intensive; and 3) optimization may fail if source code changes. First, PGO frameworks in prior studies lack flexibility. In particular, they are only applicable to specific scenarios, because their integrated algorithms for generating profile and methods for simulating real-world scenarios are limited. Therefore, when the usage scenario changes, it is difficult for the existing PGO frameworks to switch the appropriate algorithms and 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) the complete PGO process is complicated [1], it consists of many distinct phases such as instrumentation, kernel burning, profile collection, and recompiling, which are time-consuming; 2) the instrumentation phase, one of the most complex phases in PGO, includes more than 30 modifications to the Android kernel code, which are error-prone; and 3) profile collection requires extensive manual interventions such as clicking or touching the phone, which is labor-intensive.

Third, the fragmentation, diversity, and version update iterations of the Android kernels make optimizations incompatible

* Corresponding authors.

with each other. Each version of the kernel corresponds to an optimization. The optimization will become invalid when the source code changes, thereby increasing the labor consumption.

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

- (R1) **Flexibility and scalability.** The framework should be able to flexibly switch appropriate algorithms for generating profile and methods for simulating real-world scenarios in different application scenarios, and should be scalable to new algorithms and methods.
- (R2) **End-to-end.** The framework should obtain the optimized kernel image directly for the kernel source code and optimization requirements provided by the user, minimizing time overhead and manual operations.
- (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 consists of three key components: 1) a tool orchestration; 2) a novel domain-specific language (DSL) we designed and its compiler; and 3) an adapter. First, we designed a tool orchestration to integrate various algorithms for profile generation and methods for simulating real-world scenarios. On the one hand, to demonstrate the flexibility of the orchestration, it has already integrated two state-of-the-art algorithms, PGO and CSPGO (Context-Sensitive PGO)[4], and click strategies such as random clicks and traversal-based clicks. MEPOF has the flexibility to switch to the appropriate algorithm and strategy depending on the different optimization requirements. On the other hand, to make MEPOF more scalable, we have reserved specific APIs in the tool orchestration to integrate more algorithms and new simulating methods.

Second, we designed a domain-specific language dubbed G4 and its corresponding compiler G4COMPILER. The language is used to describe the requirements for the entire optimization process. The compiler takes as input the optimization description code written in G4 and outputs the required configuration files for the whole optimization process. Then, MEPOF automatically optimizes the Android kernel according to the configuration files and finally outputs an optimized kernel image. Using domain-specific language, only one input (G4 code) can obtain one output (optimized Android kernel) without additional manual intervention in the optimization process.

Third, we designed an adapter to adapt for different versions of the kernel. First, the adapter can select the desired kernel version based on the configuration. Second, the adapter can also keep 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 effectiveness, performance, and usability. First, we used manual methods and MEPOF to optimize the Android kernel respectively, and the results demonstrate that the overall performance difference does not exceed 2%, and the framework is effective. Second, we conducted four subexperiments to demonstrate that MEPOF can save up to 39.07% time compared with manual. Third, MEPOF is highly usable, it requires only one human interaction to complete the optimization, while manual optimization requires more than 30 human interactions.

**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 in this paper are as follows:

- **A PGO framework to optimize the Android kernels.** We presented MEPOF, the first end-to-end optimization framework for Android kernels using PGO. MEPOF meets three requirements : flexibility and scalability, end-to-end, and self-adaptiveness.
- **A prototype based on MEPOF.** We implemented a prototype of MEPOF consisting of a novel domain-specific language and its compiler, a tool orchestration and an adapter.
- **Evaluation of MEPOF.** We conducted extensive experiments to evaluate the effectiveness, performance and usability of MEPOF. The experimental results showed that MEPOF is effective, efficient and easy to use.

**Outline.** Section II presents the background of PGO and the challenges in applying it to Android kernels. Section III discusses the overall design of MEPOF, and Section IV presents the software prototype implementation. Section V presents the evaluations we conducted. Section VI discusses the limitations and 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). Additionally, we present 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 the *profile*.

**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 optimization goal-dependent. For example, to perform loop optimizations[8], the compiler inserts counting code at branch

points in the target program being optimized, so that the execution frequencies of each branch will be counted and recorded when the target program executes. The instrumented program will then be compiled into binaries for subsequent processing. Second, in the profile generation phase, the instrumented binaries are executed to generate runtime information. The profile contains important information such as the execution frequency of various parts of the program. The profile is collected to be used in subsequent optimizations. Third, in the program recompilation phase, the compiler will optimize the target program by leveraging the profile generated in the above phase. For example, the compiler will inline the function when a function is called frequently, thereby reducing the number of calls of the function to improve program performance.

**PGO history.** PGO has been well studied with a long history due to its potential to leverage runtime information in guiding optimizations. Studies of PGO date back at least to the 1960s [9] [10] [11]. Recently, due to the progress of hardware support [3], PGO has been extensively studied [12] [13] and has been successfully used in optimizing a variety of real-world systems such as Chrome [14], PHP [15], .NET [16], Firefox[17], Service Mesh[18], and even Linux kernels [19]. The application of PGO to optimizing these systems brings significant performance improvements. For example, the average performance of each indicator of the .NET Core 2.0 has increased by as much as 21.33% after using PGO for optimization[20].

**Compiler support.** As a promising compiler optimization, PGO has been well supported by mainstream compilers. For example, recent releases of GCC [21] and LLVM [22] have complete support of PGO and are still developing novel features. Compared with traditional optimizations, compiler support of PGO optimization brings two grand advantages: 1) making optimization more accurate; and 2) revealing new optimization opportunities. First, PGO technology can obtain feedback information when the program is running, which cannot be obtained by 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 feedback information in the profile can trigger new optimizations. For example, the profile file contains information on the number of function executions. Using this information, the compiler can determine which functions are frequently called, trigger inline optimization to reduce the number of function calls, and finally realize program optimization.

### B. Challenges for PGO-based Android Kernel Optimization

Using PGO for the Android kernel is more complicated than the normal procedure, and the main optimization process is shown in Fig. 1. First, instrumenting and building the source code (❶). This step is mainly to add PGO instrumentation support to the kernel source code, including: adding PGO instrumentation code and storing the profile in debugfs. In addition, it is necessary to prevent certain files from being

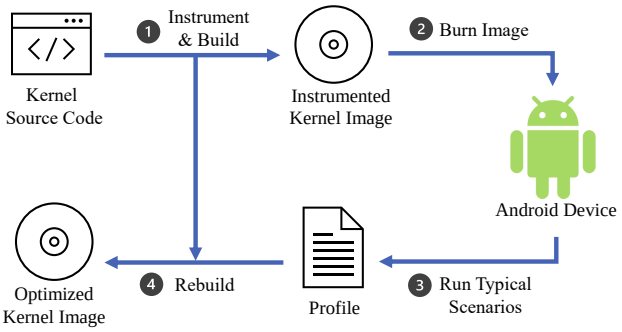



Fig. 1: The Workflow of PGO for the Android Kernel

profiled to prevent kernel crashes. Then a large number of modifications to the source code are needed, which is labor intensive. Second, burning the instrumentation kernel image into the experimental phone (❷).

Third, running typical scenarios to collect profile (❸). We can only run a set of the most commonly used apps to simulate real-world input to the kernel, since the kernel cannot run independently of the operating system. Fourth, the profile is used for recompilation (❹). In this step, it is necessary to select appropriate optimization options during compilation because not all optimizations triggered by PGO apply to the Android kernel.

The profile will become invalid when the kernel source code undergoes major changes. The profile stores information such as the execution times of function statements and branch execution probability. When the source code changes, the information recorded in the old profile cannot be used for optimization in the compilation phase, and the optimization effect will decrease or even negative optimization will occur.

## III. Design

In this section, we present the design of the MePoF in detail. We first discuss the architecture of MePoF (Section III-A) and then present each component, including the G4 domain-specific language, the G4Compiler compiler, tool orchestration, adapter, and evaluation.

### A. Architecture

The overall architecture of MePoF is presented in Fig. 2, consisting of four key components: 1) the G4 language and its compiler G4Compiler; 2) the adapter; 3) tool orchestration; and 4) evaluation. First, MePoF takes as input both the Android kernel sources and a G4 program describing optimization strategies. The Android kernel sources are either 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

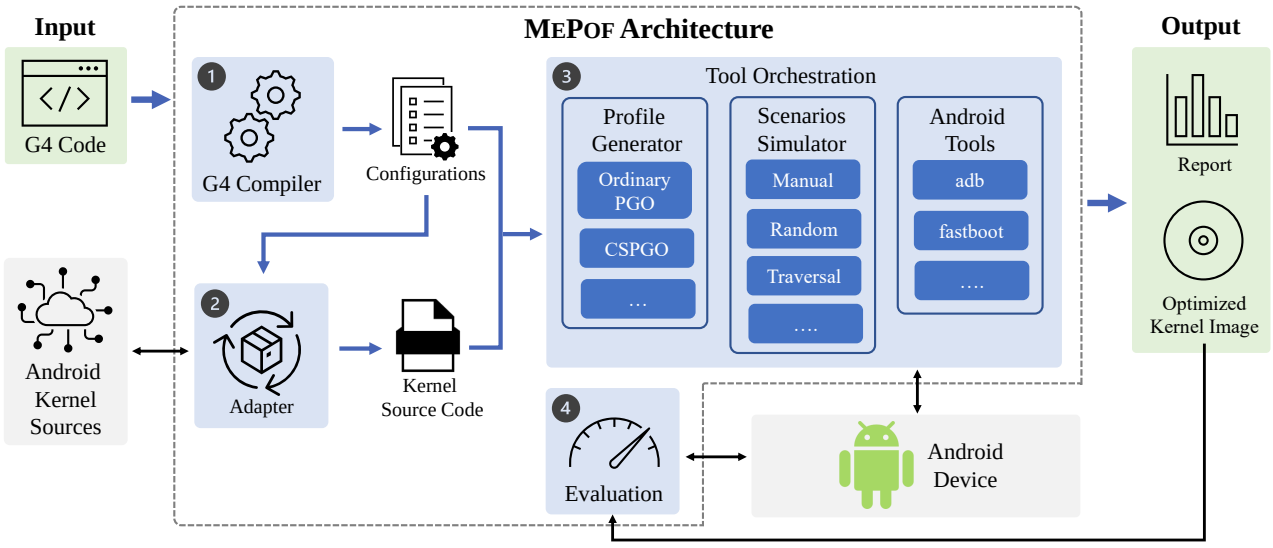



Fig. 2: The Architecture of MEPOF

(Section III-B). Second, the G4COMPILER (❶) we designed takes the G4 program as input and compiles it to configuration files employed in the 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. 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 generates an optimized Android kernel image in an automated manner. 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. This module will generate a measurement report to judge 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 PGO optimization strategies. For example, G4 should easily express various optimization algorithms, such as standard PGO and CSPGO, so that switching from one algorithm to another has zero cost. Second, the G4 language should express end-to-end decisions to reduce manual intervention. To this end, the G4 language should cover every stage of the PGO optimization processes.

With these design goals, in Fig. 3, we present the syntax of G4 via a context-free grammar. A G4 program $g$ consists of a list of statements $s$, where the notation $\vec{s}$ stands for zero or more of statements $s$ (that is, 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 $r$, or an evaluation $a$.

| | | | |
|---|---|---|---|
| Algorithm | $l$ | ::= | `pgo` \| `cspgo` \| ... |
| Click | $k$ | ::= | `manual` \| `rand` \| `trav` \| ... |
| Evaluation | $a$ | ::= | `evaluate` $img$ $\vec{metric}$ |
| Recompile | $r$ | ::= | `recomp` $path$ $p$ $\vec{op}$ |
| Adapter | $y$ | ::= | `ada` $url$ $l$ $k$ $\vec{ap}$ $\vec{op}$ |
| Merge | $m$ | ::= | `merge` $\vec{p}$ $\vec{w}$ |
| Profile | $r$ | ::= | `profile` $path$ $l$ $k$ $\vec{ap}$ |
| Optimization | $o$ | ::= | `opt` $path$ $l$ $k$ $\vec{ap}$ $\vec{op}$ |
| Statement | $s$ | ::= | $o$ \| $r$ \| $m$ \| $y$ \| $r$ \| $a$ |
| Program | $g$ | ::= | $\vec{s}$ |

Fig. 3: Syntax of G4.

All categories of statements $s$ have similar structures by starting with an operation followed by zero or more operands. In particular, optimization statement $o$ consists of an operation flag `opt` and five parameters. The parameter $path$ is a string type, which is the location of the kernel source code in the system. Parameter $l$ is the profile generation algorithm, and parameter $k$ is the method of collecting the profile. The parameter $\vec{ap}$ is a list of applications that are used for simulating real-world scenarios. The parameter $\vec{op}$ is a list of optimization options used in the recompilation phase.

A statement $r$ can generate the configurations required for program analysis, consisting of an operation flag `profile` and four parameters. A statement $m$ can generate the configurations required for merging profile files and consists of

**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
```

an operation flag `merge` and two parameters: $\vec{p}$ and $\vec{w}$. The parameter $\vec{p}$ is a list of profile files, and $\vec{w}$ is a corresponding set of proportions used for merging profile files.

A statement $y$ can generate the configurations required for automatically optimizing the specified kernel version, and consists of an operation flag `ada` and five parameters. Among them, the parameter $url$ is a string type, which is the website address where the specified version kernel updates. A statement $r$ can generate the configurations required for recompilation and consists of an operation flag `recomp` and three parameters. Among them, the parameter $p$ is the profile file that is used to guide recompilation optimization. A statement $a$ can generate the configurations required for evaluating the kernel performance and consists of an operation flag `evaluate` and two parameters. The parameter $img$ is a string type, which is the location of the kernel image. The parameter $\vec{metrics}$ is a list of performance indicators that need to be evaluated.

### C. The G4COMPILER Compiler

The G4COMPILER compiler takes as input a G4 program and compiles it to optimization configuration files. G4COMPILER follows a modular design, consisting of three key steps, as Algorithm 1 presents: 1) the front-end; 2) the abstract syntax trees; and 3) the code generator.

First, the front-end of G4COMPILER 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 such as lex[23] and bison[24] 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 G4COMPILER 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 in AST to ensure that the output is a legal configuration. For example, for the input $\vec{op}$, the user is required to add a space symbol ";" between two optimization options, and a terminal symbol "." to represent the end.

Third, the G4COMPILER compiler generates optimization configuration files from the constructed abstract syntax tree, by a postorder traversal.

### D. Tool Orchestration

Fig. 2 presents the architecture of the tool orchestration module, which 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 integrates the algorithms. To enhance the flexibility of the framework, we have reserved APIs for other algorithms. It only needs to call the corresponding API without extensive modification of the framework source code when a new algorithm is connected to the framework in the future. For example, if we want to use the CSPGO algorithm to obtain the profile, we only need to change the parameter `pgo` to `cspgo` in the G4 code.

Second, the Android kernel $k$ is instrumented according to the profile strategy $p$, obtaining a new kernel $k'$. In this step, the kernel compilation script needs to be modified to add support for instrumentation. It is worth noting that performing global instrumentation on the kernel source code may cause a kernel crash. Therefore, it is necessary to exclude modules that may cause errors. Then, the instructed 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 profile $r$, according to the simulations $s$ and Android tool configuration $t$. The Android devices not only include physical devices such mobile phones and tablets, but also virtual simulators such as Bluestacks[25] and Virtualbox[26]. To generate and collect runtime profile, the instrumented Android kernel image $i$ is first flushed to the target Android devices via specific tools such as adb[27] and fastboot[28]. Then, the target device is rebooted and benchmarks are executed on this instrumented kernel $i$, with simulation strategy $s$ and 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 example, the click method based on traversal can click each button of the app in a certain order. This method can ensure that the click sequence of each simulation is consistent and can cover all the buttons of the program.

### E. Adapter

Adapter is utilized to achieve the self-adaptiveness goal. First, in order to address optimization incompatibility caused by Android kernel fragmentation and diversity, we have built in multiple versions of kernel source code into the adapter. The module automatically matches the corresponding version based on the generated configurations. Second, in order to compensate for optimization invalid caused by kernel updates,

**Algorithm 2** : Orchestration Algorithm

**Input:** $c$: the configurations; $k$: 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
```

the adapter can track the kernel release website and automatically download the latest source code when a new version is released, and trigger an automated optimization process.

### F. Evaluation

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

Evaluation module can automatically evaluate the performance of the kernel with specific metrics. Although some benchmarks include Antutu[29] and Geekbench[30]. et al. can evaluate the performance of Android phones, they are CPU/GPU intensive, and mainly used to test the performance of the hardware. To evaluate the performance of the Android kernel, we refer to existing research on evaluating Android kernel performance [1] and select five suitable performance indicators.

To comprehensively measure the system performance of the kernel, this module mainly includes the following five indicators: system call, system call overhead[31], process creation rate, pipe throughput, and file copy rate. First, the system call is the bridge between the user state and the kernel state, and `execl()` is a system call function commonly used, so we use the `execl` throughput to evaluate the performance of the system call. Second, the system call overhead also affects the performance of the kernel. And we can record the time when the system enters and leaves the kernel mode to evaluate the overhead of the system call when the `getpid()` function (a system call function) is executed. Third, the kernel mainly consists of: process management, interprocess communication (IPC), file system and storage system, etc. Therefore, we use process creation speed to measure process performance and pipe throughput to measure interprocess communication performance. File copy operation involves the creation, reading and writing of files, and is closely related to the file system and storage system. So we use the file copy rate to measure the performance of file system and storage system.

To use the evaluation module to evaluate the optimized kernel performance, the evaluation statement $a$ in the G4 language needs to be used. The first parameter $img$ of this statement is the location of the kernel image, and the second parameter $\vec{metrics}$ is the performance index of this evaluation. For example, in the statement `evaluate /sys/out/opt.img file;pipe.`, the MEPOF will evaluate the file system performance and pipeline performance of the kernel image whose address is `/sys/out/opt.img`.

### G. Android Image and Report Generation

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

## IV. IMPLEMENTATION

We have implemented a software prototype for MEPOF. We used a handwritten strategy to implement G4COMPILER, including the scanner, parser, abstract syntax trees, and code generator. We use the `diff` tool [32] to generate the patch for ordinary PGO instrumentation and CSPGO instrumentation to implement automatic instrumentation. The orchestration module is implemented using a Python script to drive the whole optimization process. We used the Monkey tool [33] for random clicking and the Droidbot[34] as the method of traversed click to simulate real-world scenarios. We used Android platform tools including adb and fastboot, to transfer the files and burn the kernel image to experimental phones. We used Python crawler technology to monitor the kernel version release website. If the version is updated, the framework will use the downloader to automatically download the kernel source code and change the automatic optimization flag 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 experiments (V-A). Next, we evaluated MEPOF in terms of effectiveness, performance, and usability, respectively (Section V-D to V-F).

### A. Research Questions

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

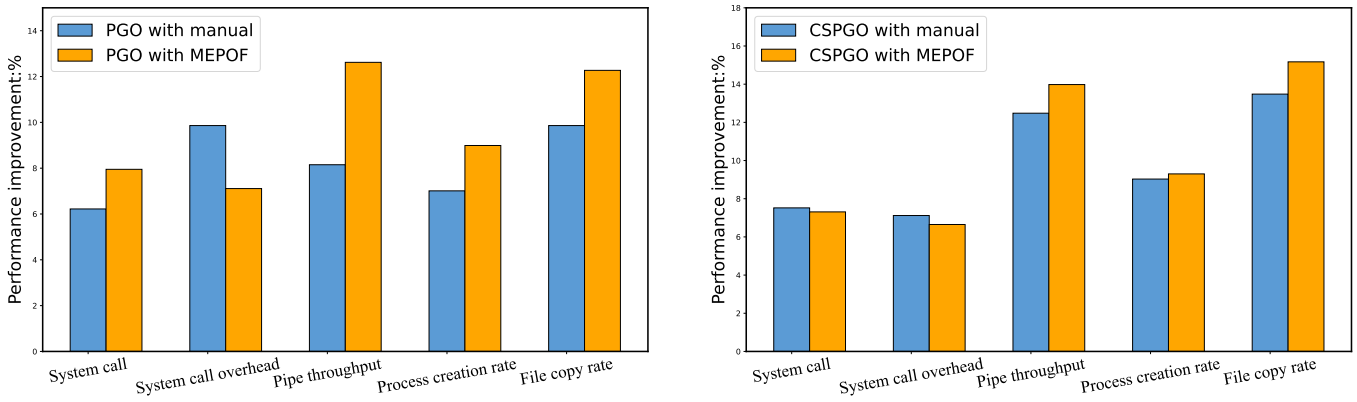
**RQ1: Effectiveness.** Is MEPOF effective in producing optimized kernels with significantly improved performance?

**RQ2: Performance.** Can MEPOF improve the performance of PGO-based optimized 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 operation system of the server is Ubuntu 20.04, the Android environment is the OP8350R project, Clang version 11.0.1, and the Android kernel is msm-5.4, which is based on Linux kernel 5.4. In addition, we use Oneplus 9Pro 5G as an experimental phone.




(a) The effectiveness of MEPOF using PGO.

(b) The effectiveness of MEPOF using CSPGO.

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

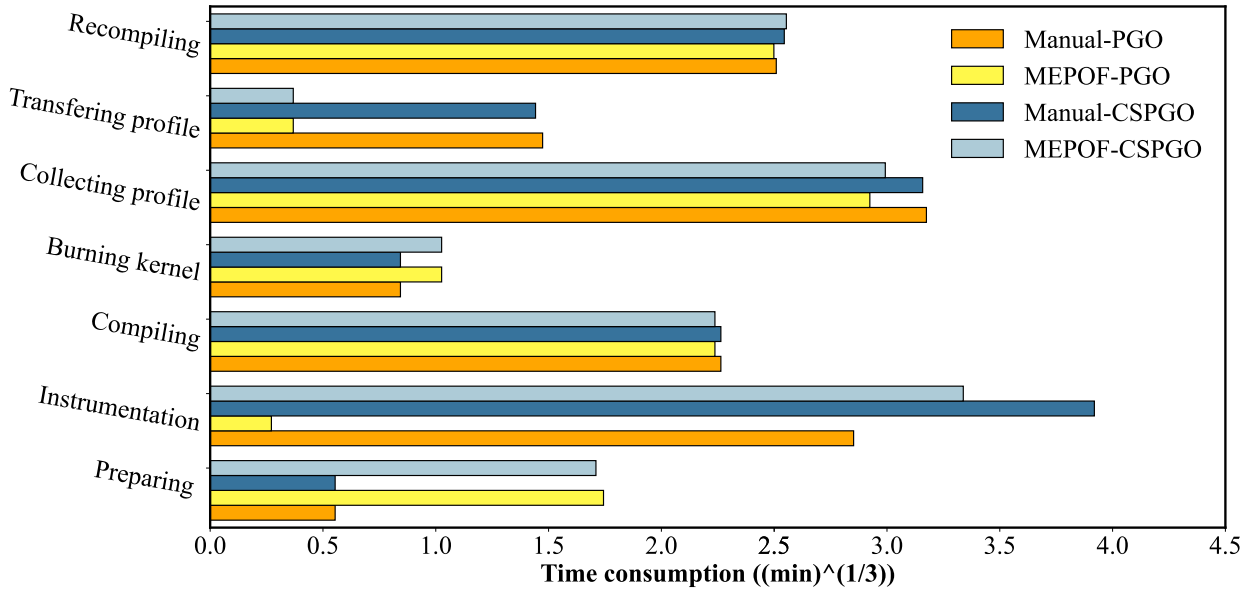



Fig. 5: The performance of MEPOF.

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

We manually constructed a micro-benchmark, including a total of 8 test cases, which can be divided into five categories, which are used to test the five performance indicators mentioned above (III-F): 1) system call performance test case by counting the execution times of the `execl()` function per second; 2) system call overhead test case by recording the time that it takes for the system to enter and leave the kernel state; 3) pipe transmission performance test cases including the number of times that 128bits data can be written and read to the pipe within one second and the number of times that 512bits data can be written and read to the pipe within one second; 4) process creation performance test case by counting the number of times a process is created and destroyed within one second; and 5) file copy rate test cases including the number of characters of a 512B file transferred within one second using a 128B buffer, the number of characters transferred within a second of a 1024B file using a 256B buffer, and the transfer of 1KB within one second using an 8KB buffer The number of characters in the file.

### D. RQ1: Effectiveness

To answer **RQ1** by demonstrating the effectiveness of MEPOF, we use manual methods and MEPOF to optimize the Android kernel, respectively, burn these optimized kernels into the same experimental phone and test their performance. The experiment was carried out 5 times, and the final experimental results were averaged.

First, we perform normal PGO to optimize the Android kernel using manual and MEPOF, respectively, and then perform performance tests on the two optimized kernels. The test results are shown in Fig. 4a. Taking the unoptimized kernel performance as the baseline, the vertical axis in the figure indicates the percentage of performance improvement. The results show that using the manually optimized Android kernel has only a slight advantage in the system call overhead, but has a slight disadvantage in the other four indicates. Among them, for the indicator of pipe throughput, the MEPOF is 4.47% higher than manual. In terms of average performance improvement, manual improved by 8.02%, MEPOF improved by 9.39%, and the two performance improvements are close.

Second, we perform CSPGO to optimize the Android kernel using manual and MEPOF, respectively, and perform performance tests on the two optimized kernels. The test results are shown in Fig. 4b. In terms of average performance improvement, manual improved by 9.13%, MEPOF improved by 10.1%, and the two performance improvements are close.

In summary, the performance of the kernel optimized using MEPOF is essentially equivalent to that of the manual optimization, and therefore MEPOF is effective.

### *E. RQ2: Performance*

To answer **RQ2** by demonstrating the performance of MEPOF, we use four methods for optimizing the Android kernel to explore whether MEPOF is more time saving than manual using the same optimization strategy.

In this experiment, we, respectively, used PGO manually, CSPGO manually, PGO in MEPOF, and CSPGO in MEPOF to optimize the Android kernel and recorded the time spent on each stage of the optimization process. In order to more accurately grasp the time consumption of each link in the PGO process, we further break down the complete process of PGO for Android kernel optimization into 7 stages: 1) the pre-optimization preparation; 2) instrumentation; 3) compiling instrumented kernel source code; 4) burning instrumented kernel image into experimental phone and reboot the phone; 5) collecting profile by running typical scenarios; 6) transferring profile and converting profile format; and 7) recompiling kernel source code using profile. Each sub-experiment was performed 5 times, and the time spent in each stage was averaged. And we performed root sign processing three times on the time consumption of each stage to facilitate data comparison. The experimental results are shown in Fig. 5.

The results show that the MEPOF has significantly improved efficiency in the stages of instrumentation and profile transfer. When using CSPGO to optimize the Android kernel, the instrumentation phase takes 60.2 minutes to manual, while it takes 37.3 minutes to MEPOF, and the time saving is 22.9 minutes. On the whole, MEPOF can improve the efficiency by 39.07% compared with manual when using PGO; MEPOF can improve the efficiency by 20.69% compared with manual when using CSPGO. Therefore, it is more efficient to use MEPOF to optimize the Android kernel than to do it manually.

TABLE I: The results of manual interventions using manual methods and MEPOF

| Phases | Manual | MEPOF | interventions |
|---|---|---|---|
| Preparing | • | • | - |
| Instrumentation | • | ○ | 1241 lines of code |
| Compiling | • | ○ | 3 instructions input |
| Burning kernel | • | ○ | 2 instructions input |
| Collecting profile | • | ○ | 1+n clicks |
| Transfering profile | • | ○ | 2 instructions input |
| Recompiling | • | ○ | 3 instructions input |

### *F. RQ3: Usability*

To answer **RQ3** by demonstrating the usability of MEPOF, we counted the manual interventions of manual methods using PGO and methods of using PGO in MEPOF.

In this experiment, manual intervention includes not only the input of instructions, but also source code changes, touch screen operations of mobile phones, etc. To understand the degree of human intervention more intuitively, we performed separate statistics on each stage of the entire optimization process, and the statistical results are shown in TABLE I. In this table, symbol • indicates that the process requires manual intervention, and symbol ○ indicates that no manual intervention is needed.

The results show that the process of using MEPOF to optimize the Android kernel requires certain operations before optimization, and the entire optimization process does not require additional manual intervention, while using manual methods not only needs to modify the source code to increase support for instrumentation, but also requires more input instructions to complete a large number of manual operations, such as kernel compilation, burning, and profile collection. Therefore, MEPOF has usability characteristics.

## 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 toward designing and implementing a practical PGO-based optimization framework for Android kernels.

**Profile generation techniques.** PGO-based optimizations rely on the generation of profile, and prior studies have proposed two categories of techniques for generating profile: 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 is promising to adopt the hardware sampling technique for generating profile, as it is more lightweight without the effort and cost of source instrumentation. However, hardware sampling is a relatively new hardware technology that is only available on recent ARM64 CPUs with an embedded trace macrocell (ETM) [35] feature (note that the different but equivalent technology on x86_64 CPUs is the LBR function [36]), thus, to the best of our knowledge, few platforms have this hardware feature shipped. In the future, 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 the instrumentation technique, as Android kernels can always be sampled whether 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 their effectiveness. In addition, recent studies have proposed other simulation techniques (e.g., deep learning-based simulations [37]), which have the potential to generate more accurate profile. In the future, we plan to investigate the deep learning-based approach for profile generation. 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 these techniques into MEPOF.

**Optimization options.** Optimization options and their correction combinations are indispensable to generate efficient Android kernels from profile. 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 optimization 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 [38] [39], 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 [40]). In the future, 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 easily specify strategies in PGO-based optimizations. The G4 programs are then compiled, by G4COMPILER, into configurations for subsequent processing. 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 the future, 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 [41], and the corresponding G4 programs can be synthesized automatically. Furthermore, it is also promising to synthesize G4COMPILER in an automated manner similar to JitSynth [42] did, and we also leave it for 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 general software systems in the wild. However, the work in this paper represents a novel contribution to this field.

**Profile generation.** Profile generation has been widely studied, and two approaches have been 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 not context sensitive to address this limitation, Xu et al. [4] proposed CSPGO, a novel instrumentation to achieve better optimization. Ellis et al. [43] proposed a lightweight PGO (IRPGO) to reduce runtime overhead. This is a lightweight instrumentation algorithm, which greatly reduces the expansion of the instrumentation program, making it more suitable for mobile devices. Among the sampling algorithms, Chen et al. [14] proposed the AutoFDO algorithm, which generates profile by mainly collecting the last branch record (LBR). They showed that the runtime overhead is reduced significantly, as the source code is not instrumented. Diego Novillo [44] proposed SamplePGO, which uses an external sampling analyzer to obtain a profile. Since no instrumentation is used, the algorithm has almost negligible runtime overhead. Although the sampling method can reduce runtime overhead, the profile obtained by the sampling method is not as accurate as that obtained by the instrumentation method. Since this framework can reduce labor, we mainly integrate the instrumentation algorithm to obtain the profile.

**PGO-based optimizations.** PGO-based optimizations have been studied extensively. Wang et al. [12] used PGO for dataflow prediction. Huang et al. [45] used PGO for indirect branches in a binary translator. Homescu et al. [13] used PGO technology to solve the huge performance overhead caused by using software diversity to defend against code reuse attacks, and improved its performance on the basis of improving software security. However, optimizing large software with PGO is time-consuming, labor-intensive and boring. Therefore, the framework proposed in this paper can effectively address these limitations.

## 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 designed three core components, including a tool orchestration to flexibly switch algorithms and strategies for different usage scenarios, a novel domain-specific language G4 and its compiler to specify optimization strategies, and translate them into the necessary configuration files in an automated manner, 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 toward applying PGO-based optimizations to Android kernels, thus making Android,

the most pervasive kernel for today's mobile computing, more efficient and competitive.

## ACKNOWLEDGMENTS

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 documentation," 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] "Pgo on service mesh," https://zhuanlan.zhihu.com/p/496544763.

[19] 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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[37] 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.

[38] 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.

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

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

[41] 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.

[42] 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.

[43] "Irpgo," https://discourse.llvm.org/t/instrprofiling-lightweight-instrumen tation/59113.

[44] 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.

[45] 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.