Calibro: Compilation-Assisted Linking-Time Binary Code Outlining for Code Size Reduction in Android Applications

Zhanhao Liang*
Wuhan University
Wuhan, China
zhliang@whu.edu.cn

Hanming Sun*
Wuhan University
Wuhan, China
hanmingsun@whu.edu.cn

Wenhan Shang
Wuhan University
Wuhan, China
whu_swh@whu.edu.cn

Mengting Yuan
Wuhan University
Wuhan, China
ymt@whu.edu.cn

Jingqin Fu
Wuhan Broadcasting and Television Station
Wuhan, China
fudaisy11@163.com

Jiang Ma†
Guangdong OPPO Mobile Telecommunications Crop., Ltd.
Dongguan, China
majiang@oppo.com

Chun Jason Xue
MBZUAI
Masdar, United Arab Emirates
jason.xue@mbzuai.ac.ae

Qingan Li†
Wuhan University
Wuhan, China
qingan@whu.edu.cn

Abstract

Recent Android systems have employed pre-compilation technology to boost app launch speed and runtime performance. However, this generates large OAT files that overconsume scarce memory and storage resources in mobile devices. This paper conducts an evaluation of code redundancy in popular production android applications and observes that the code redundancy is up to 25%. To reduce the code size via redundancy elimination, this paper proposes Calibro, a compilation-assisted linking-time binary code outlining method. Calibro consists of two parts, the Compilation-Time code Outlining (CTO) and the Linking-Time Binary code Outlining (LTBO) with information collected at compilation-time. Additionally, a paralleled suffix tree method is proposed to reduce the building time overhead, and a hot function filtering method is proposed to effectively mitigate run-time performance degradation caused by code outlining. Experimental results show that compared to the baseline (the original AOSP version with all available code size optimization enabled), the proposed approach reduces code size in Android applications by more than 15.19% on average, with negligible runtime performance degradation and tolerable building time overhead. Hence the proposed code outlining approach is promising for production deployment.

CCS Concepts: • **Software and its engineering** → **Compilers**.

Keywords: Code outlining, Code size, Android systems

ACM Reference Format:
Zhanhao Liang, Hanming Sun, Wenhan Shang, Mengting Yuan, Jingqin Fu, Jiang Ma, Chun Jason Xue, and Qingan Li. 2025. Calibro: Compilation-Assisted Linking-Time Binary Code Outlining for Code Size Reduction in Android Applications. In *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization (CGO '25), March 01–05, 2025, Las Vegas, NV, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3696443.3708955

*The first two authors contributed equally to this research.
†Corresponding author.

CGO '25, March 01–05, 2025, Las Vegas, NV, USA

ACM ISBN 979-8-4007-1275-3/25/03
https://doi.org/10.1145/3696443.3708955

1 Introduction

As the importance of mobile applications in daily life continues to grow, modern businesses are increasingly reliant on mobile apps for their business models [6]. This rapid growth in business demand has heightened the requirements for mobile app functionality, leading to continuous expansion and enhancement of features, which in turn causes code size inflation. Larger applications require more resources for downloading, installation, storage, and execution. Moreover,

many app download platforms impose size restrictions. For instance, from August 2021, new apps are required to publish with the Android App Bundle on Google Play [22] and Google Play enforces a compressed download size restriction of 200 MB for apps published with app bundles [23]. Research [6] indicates that when app size approaches or exceeds these limits, user satisfaction decreases, resulting in significant drops in App registrations, installations, and initial bookings, thus causing losses for mobile app developers. Therefore, reducing the code size of mobile applications has become a critical objective for both mobile app developers and device manufacturers.

Android is a widely used mobile platform. As of September 2024, Android holds 71.85% of the global mobile market share [32]. In the Android system, most applications are written in JAVA or Kotlin and run on the Android virtual machine. The method of interpreted execution significantly slows down both the startup speed and the runtime efficiency of programs. Consequently, starting with Android 7.0 (Nougat), Ahead-Of-Time (AOT) compilation is supported to compile applications into binary code that runs directly on the device and stored in the form of OAT files on Android devices. These OAT files often constitute a significant portion of the application size. Additionally, compared to traditional compilers like GCC and LLVM, the code-size-oriented optimizations of Android's compilers are relatively weak, resulting in binary code with a considerable amount of unnecessary or redundant code. According to this paper's analysis, the code redundancy in OAT files is estimated to be more than 25%, as illustrated in Table 1. Therefore, OAT files are critical to code size reduction for Android applications.

Existing work proposed code outlining-based methods to reduce the code size either for general applications by implementing the methods in the LLVM compiler infrastructure [21] [25], or specifically for iOS applications [6] [19]. However, it cannot be directly applied to Android applications due to the following challenges.

- There is a lack of relevant work on Android systems. In Android systems, OAT files are special ELF files, containing a part of Android-specific content. Simply applying code outlining can lead to a waste of optimization opportunities, and even severe correctness issues due to these differences;
- Unlike iOS's build pipeline, Android's build pipeline doesn't support a machine-level intermediate representation (IR). As a result, the whole program outlining could only be conducted at link-time on the binary code, rather than the assembly code as in previous work [6]. Outlining on the binary code is much more challenging than outlining on the assembly code since it involves the work of disassembling and binary rewriting;
- Code outlining causes performance degradation. Code outlining always introduces additional execution of call and return instructions, which is unfriendly to both the CPU pipeline and code cache, thus leading to inevitable performance degradation. Meanwhile, this optimization also causes building overhead.

To overcome these challenges, this paper proposes Calibro. We analyzed the features of binary code redundancy in OAT files and conducted a compilation-time code outlining based on the result. Furthermore, in the compilation time, we collected essential information like PC-relative addressing instructions' offsets and targets that will assist the Link-Time Binary code outlining just followed. In addition, we proposed a paralleled suffix tree method which significantly reduces the building time overhead. To mitigate the performance degradation caused by code outlining, we proposed a hot function filtering method and and successfully reduced the overhead to less than 1%.

Experimental results show that Calibro can reduce code size in Android applications by more than 15%, with negligible runtime performance degradation and tolerable building overhead, which enables the code outlining approach to be promising for the production environment.

The main contributions of this work include:

- This work analyzes the code redundancy in Android applications. Based on the evaluation, we find several observations, which motivate this work to reduce the code size in Android applications;
- This work proposes a compilation-time code outlining method to optimize the top repetitive code patterns specific to the Android Runtime (ART). This method can reduce the ART-specific code redundancy in a much more lightweight way than a general code outlining work;
- This work proposes a link-time binary code outlining method with the information collected at compilation-time. With this useful information, the binary code outlining work is exempt from bothering the headache-inducing work of disassembling and binary rewriting;
- This work proposes a hot function filtering method to mitigate the runtime performance degradation due to code outlining. In addition, this work proposes a paralleled suffix tree method to speed up the binary code outlining work;
- This work conducts a comprehensive experimental evaluation of the proposed method, and the results show that the proposed approach can significantly reduce the code size of Android applications while introducing negligible performance degradation and tolerable building overhead. It demonstrates that the proposed code outlining method is promising for the production environment.

2 Background and Motivations

2.1 Background

2.1.1 Code Outlining. Code outlining is widely used for code size reduction. The basic workflow of code outlining is, identifying repetitive code sequences within the code, outlining this sequence into functions, and subsequently replacing all instances of this sequence with calls to the newly created outlined function. This method can generally be divided into two main steps: Code redundancy detection and elimination.

Code Redundancy Detection: Code outlining methods commonly utilize suffix tree-based redundancy detection schemes to identify all redundant code segments within the code [6, 18, 21, 25]. Initially, the whole code sequence of an application is transformed into an unsigned integer sequence through instruction mapping or instruction hashing[19]. A suffix tree, is subsequently built using the sequence, and the non-leaf nodes of the suffix tree are traversed to find all recurring code sequences. The suffix tree will be discussed in detail later.

Code Redundancy Elimination: Once duplicate instruction sequences are identified, these sequences are outlined into functions, and all identical sequences are replaced with calls to these outlined functions.

In recent years, outline methods have garnered extensive attention from both the industry and academia, finding significant applications in the mobile domain. LLVM, a famous open source complier framework, has implemented code outlining in both IR level[21] and machine IR level [25]. Building on this, Uber's research [6] modified the compilation pipeline to merge multiple IR files, expanding the optimization scope from a single module to the whole program. This approach, through multiple rounds of outlining, significantly reduce the code size. Uber's study was the first to apply code outlining to commercial iOS applications, demonstrating its effectiveness in the mobile domain. Subsequently, Meta's research [18] employed code instrumentation and dual code generation to mitigate the overhead in compilation time and performance degradation caused by code outlining. More recently, ByteDance's research [19] further reduced the code size of native iOS applications by analyzing and rewriting binaries during the linking process, leveraging outlining to handle redundant parts. However, most of these work focus on reducing code size of iOS applications, cannot be directly applied to the Android system due to the significant challenges as stated in Section 1.

2.1.2 Suffix Tree. A *suffix tree* is a data structure designed for efficiently storing and retrieving all suffixes of a string set, and it is a special type of trie. In simple terms, a suffix tree built from a sequence is a compressed trie built from all the suffixes of that sequence.

Figure 1 shows an example suffix tree from the string "banana". This string has a set of seven suffix substrings: "banana$", "anana$", "nana$", "ana$", "na$", "a$", "$" , where "$" is used to indicate the ending mark. As shown in this figure, each leaf-node represents a suffix substring which consists of the ordered characters on the path from the root node to this leaf-node. For example, the leaf-node with the label "1" represents the suffix substring "anana$".

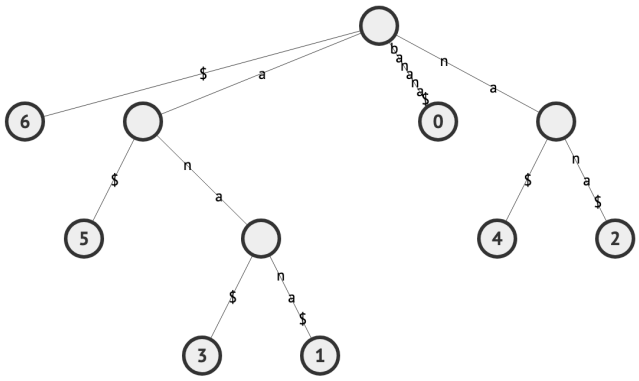

Figure 1. An example of suffix tree.

The suffix tree can be used to find repetitive substrings. Specifically, for each non-leaf node, if the number of transitively descendant leaf nodes, N, is greater than or equal to 2, the corresponding sequence is identified as repetitive, indicating that it appears N times among this application's code. Moreover, it's simple to obtain the length of each repetitive code sequence, which is the length of characters on the path from the root node to this non-leaf node.

For example, in Figure 1, the rightmost non-leaf node has two descendant leaf-nodes, i.e., the nodes with labels "4" and "2". It indicates the substring "na", which consists of characters on the path from the root node to this non-leaf node, appears twice in two suffix substrings, i.e., "na$" and "nana$", respectively.

Note that in the suffix tree there may be overlapping repetitive substrings. For example, in Figure 1, the second leftmost non-leaf node indicates the substring "ana" appears twice in suffix substrings "ana$" and "anana$" respectively. But these two substrings overlap with each other. To find the non-overlapping repetitive substrings, a small modification should be applied to selectively skip such ones.

The code redundancy detection work often employs suffix tree based methods, which primarily utilize the suffix tree to identify non-overlapping repetitive sequences among the code sequence representing the whole application, simply by viewing the sequence as a string. After that, the non-overlapping repetitive sequences can be outlined into a function to eliminate the code redundancy.

2.2 Analysis of Code Redundancy in Android Applications

We select several most popular mobile applications, mainly covering short video and social media applications, in the OPPO App Market to analyze the code redundancy in Android applications. Based on the analysis, the potential code size savings due to code outlining can be estimated. The analysis process consists of four steps as below:

(1) Mapping Binary Code into a Sequence of Unsigned Integers: Firstly, the application's binary code is disassembled and thus an instruction sequence (IS) is obtained. Each instruction is then mapped to a unique unsigned integer, generating a sequence of unsigned integers, V.

(2) Building a Suffix Tree Using the Sequence V: The sequence of unsigned integers V is used as input to build a corresponding suffix tree using the Ukkonen algorithm [36], which has the time complexity $O(n)$, where n is the length of the input string. In the suffix tree, each leaf node can be traced by the path from the root node to this node, which corresponds to a suffix sequence of V.

(3) Detecting the Repetitive Sequences: Non-leaf nodes in the suffix tree are traversed. For each non-leaf node, if the number of transitively descendant leaf nodes, N, is greater than or equal to 2, the corresponding sequence is identified as repetitive, indicating that it appears N times among this application's code.

(4) Estimating the Code Size Savings by Code Outlining: The potential code size reduction ratio by code outlining is estimated using the equations in Figure 2. Here, *Length* means the number of instructions within a repetitive sequence, *RepeatedTimes* means the number of repetitions, and 1 indicates the extra *ret* instruction to support the code outlining. As stated before, it's simple to obtain both *Length* and RepeatedTimes for each repetitive code sequence in the built suffix tree.

$$
\begin{aligned}
OriginalSize &= Length \times RepeatedTimes \\
OptimizedSize &= RepeatedTimes + 1 + Length \\
ReductionRatio &= (OriginalSize - OptimizedSize)/OriginalSize
\end{aligned}
$$

Figure 2. The benefit model for code size reduction by code outlining.

2.3 Observations and Motivations

Based on the analysis of code redundancy mentioned above, some interesting observations are found.

2.3.1 Observation 1: The Binary Code in OAT Files Contains Significant Redundancy.

Table 1 illustrates the estimated potential code size reduction for the selected popular applications. The average estimated code size reduction ratio is 25.4%, indicating substantial potential benefits from code outlining.

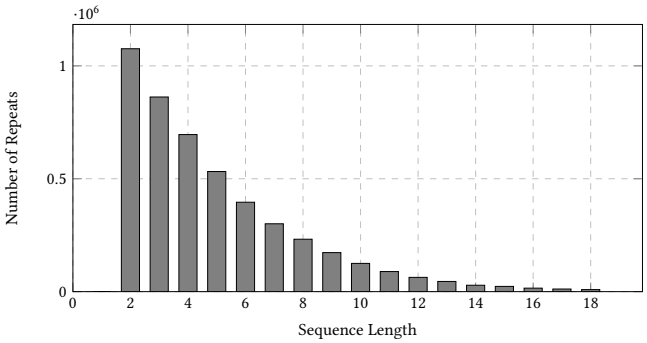

Figure 3. Sequence Length vs. Number of Repeats.

2.3.2 Observation 2: Code Sequences with High Repetition Frequencies Are Commonly Very Short.

As shown in Figure 3, most repetitive code sequences are short, and the shorter the length of the sequence, the higher the frequency of repetition. Therefore, most of repeating sequences are typically confined within a basic block. This hints that code outlining within a basic block is highly promising, given that outlining of code sequences across a basic block is rather challenging.

2.3.3 Observation 3: Several Repetitive Code Patterns Are Specific to the Android Runtime (ART).

We analyze the repetitive code sequences with the highest repetition frequency in the Wechat App in detail, and find that these repetitive sequences are highly specific to the ART features. Among these sequences, we highlight three hottest repetitive code patterns: The Java function calling pattern, the ART native function calling pattern, and the stack overflow checking pattern.

Java Function Calling Pattern: Figure 4a shows the Java function calling pattern. In ART, each Java function is represented as an internal structure instance called *ArtMethod*. To invoke a Java function, the ART firstly loads the associated *ArtMethod* into the *x0* register, and secondly loads the entry address of the callee function from the *ArtMethod* object with a fixed offset value into the *x30* register, and finally jumps to the function code by *blr x30*. The latter two steps exhibit the exactly same code sequences for multiple invocations of the same function. One instance with the offset value of 20 appears 1006k times in the Wechat App, and is ranked the most repetitive code sequence.

ART Native Function Calling Pattern: Figure 4b shows the ART native function calling pattern. The ART itself provides some native runtime functions to handle the runtime tasks like memory allocation, memory release, .etc. These native functions are preload into a memory segment, while the segment address is always stored in a thread register (*x19*), and each individual native function is addressed by this segment address plus a fixed offset. One instance of this pattern (invoking *pAllocObjectResolved*) appears 217k times

Table 1. Estimated code size reduction ratios in popular apps on OPPO App Market.

	Toutiao	Taobao	Fanqie	Meituan	Kuaishou	Wechat	AVG
Estimated reduction ratios	**25.4%**	**26.3%**	**24.5%**	**24.3%**	**27.7%**	**24.3%**	**25.4%**

in the Wechat App, and is ranked the third most repetitive code sequence.

Stack Overflow Checking Pattern: Figure 4c shows the stack overflow checking pattern. In ART, each non-leaf function should check the stack address to avoid stack overflow before any further stack manipulation, and an exception will be thrown if the checking fails. This stack overflow checking mechanism uses a fixed code sequence as illustrated in Figure 4c. This sequence appears 173k times in the Wechat App, and is ranked the second most repetitive code sequence.

These patterns are common across ART and are likely present in all Android applications. They can be identified during the compilation stage, and redundancy can be eliminated by modifying the instruction generation with code outlining.

2.4 Motivations

In the Android systems, most compilation optimizations are concentrated at the intermediate representation level, HGraph, as illustrated in Figure 5. However, much code redundancy cannot be identified on the high level of abstraction in HGraph. In addition, many optimizations are conducted per function, only eliminating inner function redundancy and unreachable code. As a result, lots of code redundancy remains in the generated binary code, as confirmed by Observation 1.

Based on Observation 1 and 2, we are motivated to reduce the OAT code size for Android applications by a whole program code outlining method. Specifically, as Android's build pipeline doesn't support a machine level IR, the whole program code outlining could only be conducted at linking-time on the binary code.

```
ldr x30, [x0, #offset]
blr x30
```

(a) The Java function calling pattern. Here *#offset* indicates a constant offset value.

```
ldr x30, [x19, #offset]
blr x30
```

(b) The ART Native Function Calling Pattern. Here *#offset* indicates a constant offset value.

```
sub x16, sp, #0x2000 (8192)
ldr wzr, [x16]
```

(c) The Stack Overflow Checking Pattern.

Figure 4. The repetitive code patterns specific to the ART.

Based on Observation 3, there are three repetitive code patterns which are common across ART and are likely present in all Android applications. This code redundancy can be reduced by lightly modifying the code generation work, which is much more lightweight than a general code outlining work as discussed later.

3 Design

This paper proposes Calibro, a compilation-assisted linking-time binary code outlining approach. Calibro consists of two parts, the Compilation-Time code Outlining (CTO) method and the Linking-Time Binary code Outlining (LTBO) method. The LTBO method firstly collects information at compilation-time, and then conducts the whole program binary code outlining at linking-time. The workflow of Calibro is illustrated in Figure 5, where the filled rectangles highlight the work proposed in this paper.

As illustrated in Figure 5, in Android systems, an application package consisting of multiple dex files is translated into binary code in the form of an OAT file by a tool called DEX2OAT. Each dex consists of multiple methods. In DEX2OAT, each method is translated into the IR called Hgraph, optimized on the IR, and then further translated into binary code, independently from other methods. Finally, all compiled methods in the form of binary code are linked into the OAT file.

The proposed approach coordinates the compilation-time work and the linking-time work. During the compilation-time, it works after the IR optimizations, firstly conducting the CTO method for the three ART specific repetitive code patterns, and then collecting useful information, which is the first part of the LTBO method. During the linking-time, it conducts the whole program binary code outlining, which is the second part of the LTBO method. Since the whole program code outlining approach may introduce issues like building overhead and runtime performance degradation, we also propose two optimization methods to address these issues.

3.1 CTO for ART Specific Repetitive Code Patterns

For the three repetitive code patterns specific to the ART as discussed in Observation 3, we propose a lightweight compilation-time code outlining method. We can outline these repetitive code patterns by modifying the code generation work in DEX2OAT.

In DEX2OAT, the code generation work traverses each IR instruction and generates corresponding binary code based on instruction templates. To outline the three specific code

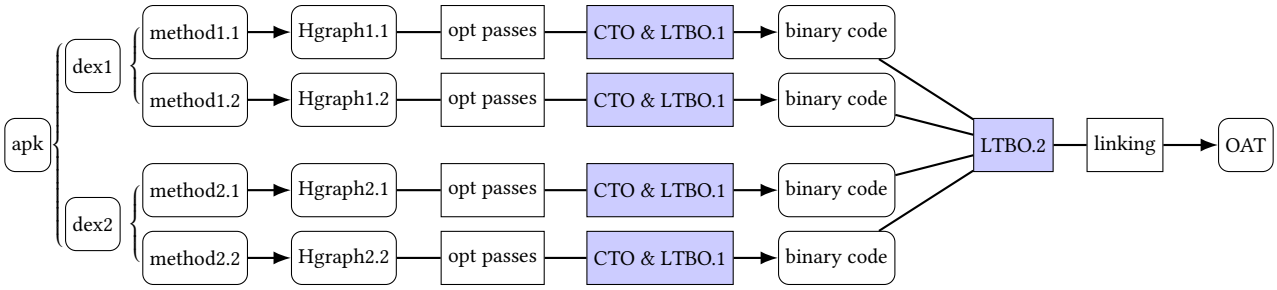

Figure 5. The workflow of the Calibro approach for Android applications.

patterns, we can simply modify their corresponding instruction templates in this way: (1) Generating the original code sequence, if it has not been generated, and storing it in a cache with a label L; otherwise, retrieve the label L of the code sequence from the cache; (2) Generating a call instruction with label L as its target.

3.2 LTBO at Compilation-time

A basic version of link-time binary code outlining works as follows.

- Disassembling the binary code into assembly code.
- Tracking PC-relative addressing instructions, like jump instructions.
- Detecting repetitive code sequences among the whole program.
- Outlining by reserving a single copy of the repetitive code sequence as a function, and replacing all occurrences with a jump or call instruction whose target is the reserved copy. This step commonly changes the relative offset between instructions, and thus may invalidate instructions of PC-relative addressing. For example, in a jump instruction, the offset relative to the PC register becomes outdated.
- Patching each PC-relative addressing instruction to update its target with the correct one. There is no need of patching the call instructions, no matter they are PC-relative addressing or not. Because the target labels of call instructions, i.e., function labels, have not been bound to addresses or offsets at this time. Instead, the later linking phase after linking-time outlining will bind function labels to addresses, and relocate the call instructions to the corresponding addresses.

This basic version of link-time binary code outlining optimization face challenges of correctness issues. The binary file sometimes embeds data like string literals or constants directly within the code for performance, which may mislead the disassembler to mistakenly decode the embedding data as instructions. In addition, since the target of an indirect jump instruction is commonly determined by address computation at runtime, at the link-time it is difficult to both evaluate its old target and modify these address computation instructions to achieve a desiring new address.

To solve these issues, we propose LTBO, a compilation-assisted link-time binary code outlining method, by recording useful information during the compilation-time and doing the whole program outlining during the linking-time. The information to collect relates to embedded data, instructions of PC-relative addressing, terminator instructions, indirect jump instructions, slowpath and Java native methods, and is expected to assist the link-time work to avoid the correctness issues above.

- Embedding data: Record the embedding data's offset and size.
- Instructions of PC-relative addressing: Record the offsets of these instructions, as well as those of their targets.
- Terminator instructions: Record the offsets of instructions terminating a basic block, such as jump and return instructions.
- Indirect jump instructions: Record a flag to mark that the owner method of indirect jump instructions is not suitable to be outlined due to correctness issues.
- Java native methods: Record a flag to mark Java native methods. These methods are not suitable to be outlined due to the difficulty of binary rewriting.
- Slowpath: Record a flag to mark the code in a slowpath. This code is commonly not frequently executed and not performance critical, and is suitable to be outlined even in hot functions.

3.3 LTBO at Linking-time

The information collected at compilation-time can be used as below.

- The embedding data can be precisely recognized to avoid disassembling.
- The methods with indirect jump instructions, as well as Java native methods, can also be precisely recognized to avoid outlining.
- The information of terminator instructions can be used to separate basic blocks.

- The information of PC-relative addressing instructions are enough for them to be patched to keep their offsets updated.

As a result, the improved, compilation-assisted linking-time outlining method can be conducted without bothering the challenging work of thorough disassembling and binary writing, consisting of four steps: Choosing candidate methods to outline, detecting repetitive code sequences, outlining the binary code, and finally patching PC-relative addressing instructions.

3.3.1 Choosing Candidate Methods to Outline. As stated before, the methods with indirect jump instructions and the Java native methods can be recognized using the information collected during compilation-time, and should be excluded from the outlining optimization. The remaining methods constitute the candidate methods to outline.

3.3.2 Detecting Repetitive Code Sequences. The repetitive code sequences are detected via a suffix tree based method, as depicted in Section 2.1.2. Note that for the outlining aim, each repetitive code sequence should be confined within a basic block and should not be ended with a terminator instruction, otherwise it is difficult to be outlined into a single-entry-single-exit function. To enforce this condition, each terminator instruction collected at compilation-time will be mapped into a unique separator number. When building the suffix tree, the separator number terminates a sequence, thus confining each repetitive code sequence in the suffix tree within a basic block. Note that the instruction mapping is simpler right now. In fact, the encoding number of each instruction can be directly used in the sequence, except that all terminator instructions should be mapped to a single unique separator number.

3.3.3 Outlining the Binary Code. As stated in Section 2.1, for each repetitive sequence in the suffix tree, it is simple to know its length and the number of repeats. Based on this information and the benefit model illustrated in Figure 2, we can evaluate whether it is worthwhile to outline a repetitive code sequence into a function. Moreover, based on this information and the benefit model, we can also choose the sequence with larger benefit among multiple overlapping ones to outline.

Once it's beneficial to outline a code sequence, we can create a outlined function by reserving the original repetitive code sequence, plus an extra instruction jumping to the return address, as illustrated below. Then, we replace all occurrences with a call instruction (the *bl* instruction in ARM), whose target is the outlined function.

An example is shown in Table 2, where code 1 shows the original code sequence, with two underlined instructions to be outlined. The outlined function is shown in code 2, with an additional *br x30* instruction to return to the calling site. After outlining, in the original code sequence, the outlined two underlined instructions are replaced with a single *bl* instruction to call the outlined function, as shown in code 3.

3.3.4 Patching PC-relative Addressing Instructions. The outlining step commonly changes the code size as well as the relative offsets between instructions, and thus may invalidate instructions of PC-relative addressing. In AArch64, the PC-relative addressing instructions include *b*, *bl*, *cbz*, *cbnz*, *tbz*, *tbnz*, *adr*, *adrp* and *ldr*. To patch these instructions in order to update the correct target, we can use the information collected at compilation-time: Each PC-relative addressing instruction and its target. Assume that after outlining, the target is placed at a new address, then we can patch this instruction with the new target address.

As shown in the example in Table 2, after outlining, the target of the first conditional jump instruction *cbz* is placed in a new address 0x138328, and the offset in the *cbz* instruction is outdated. We should patch this instruction to update its offset from 0xc to 0x8, updating its target address with the correct one, as shown in code 4.

3.4 Optimization

As stated above, the LTBO method involves lots of efforts. The existing Ukkonen algorithm can be used to build a suffix tree in linear time to the number of machine instructions,

Table 2. An exmaple of code outlining and patching.

```
// Code 1: Original Code Sequence
0x138320: cbz w0, #+0xc (addr 0x13832c)
0x138324: ldr w2, [x0]
0x138328: cmp w2, w1
0x13832c: mov x3, x4
0x138330: ldr x3,[w0]

// Code 2: Outlined Function
0x145224 <MethodOutliner>:
0x145224: ldr w2, [x0]
0x145228: cmp w2, w1
0x14522c: br x30

// Code 3: Replace the Original Code Sequence with
// Outdated Offset
0x138320: cbz w0, #+0xc (addr 0x13832c)
0x138324: bl 145224 <MethodOutliner>
0x138328: mov x3, x4
0x13832c: ldr x3,[w0]

// Code 4: Patch the Original Code Sequence with
// Updated Offset
0x138320: cbz w0, #+0x8 (addr 0x138328)
0x138324: bl 145224 <MethodOutliner>
0x138328: mov x3, x4
0x13832c: ldr x3,[w0]
```

which can be very large for popular applications on Android, often reaching millions. In addition, the global suffix tree requires a large memory footprint. As a result, the time to build and search a global suffix tree may take up a large portion of the building time, as discussed in Section 4.

Meanwhile, the code outlining method potentially impact the runtime performance, due to the introduced additional call instructions, which are unfriendly to both CPU pipeline and code cache, leading to performance degradation.

This paper propose to speed up the code outlining with paralleled suffix trees, and to mitigate the performance degradation due to code outlining by hot function filtering.

3.4.1 Speeding Up the Code Outlining with Paralleled Suffix Trees.

To speed up the outlining, we propose to build the suffix tree in parallel. Firstly, we simply partition the candidate methods into K groups evenly in terms of method numbers, where the choice of K depends on the parallel capability of hardware. Considering the time overhead, we choose a simple and random partition instead of clustering similar method into a group. Secondly, we build a suffix tree for each group in parallel. Thirdly, we detect repetitive code sequences, outline the binary code and patch PC-relative addressing instructions per suffix tree in parallel.

The benefits of outlining in parallel is two-fold. Firstly, the capability of hardware parallelism can be exploited to speed up the computation work of outlining. Secondly, the outlining speed also benefits from the memory efficiency due to smaller working set. A global suffix tree often involves a very large working set, which places big burden on memory footprint and may deteriorate system performance. Under this situation, partitioning a single large suffix tree into multiple smaller suffix tree could improve the memory efficiency. The disadvantage is that the proposed paralleled outlining method may negatively impact the effectiveness of code size reduction, since the code outlining is local to each small suffix tree, may ignoring the common repetitive sequences across multiple suffix trees. However, the impact on the effectiveness of code size reduction is tolerable as illustrated in Table 4.

3.4.2 Mitigating the Performance Degradation Due to Code Outlining by Hot Function Filtering.

It is observed that, the outlining of hot code can lead to frequently executing of jump instructions, resulting in significant performance degradation, whereas the outlining of cold code has negligible performance impact. In addition, at the instruction level, some code is used to handle exceptional cases, known as *slowpath*, and is commonly cold even within hot functions.

Based on these observations, we propose to mitigate the performance degradation due to code outlining by hot function filtering. The workflow is illustrated in Figure 6. It collects the runtime data for each application using *simpleperf* [3]. This data primarily includes the execution time of each function. The information will guide the next building, where the code outlining will be applied only to cold methods and *slowpath* of hot functions. In evaluation, we sort the functions by their execution time and choose the set of top functions that account for 80% of the total execution time as hot functions to be filtered. The effectiveness of the hot function filtering performance optimization will be demonstrated in Section 4.

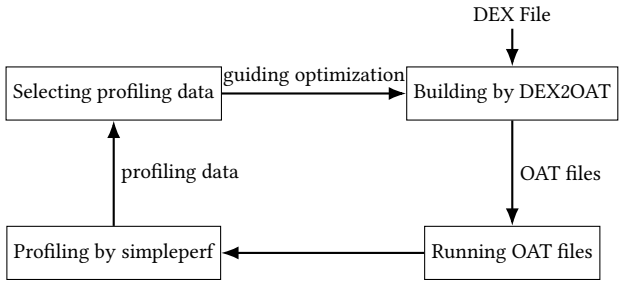

Figure 6. The workflow of the hot function filtering optimization.

3.5 Implementation Discussions

The proposed design is implemented in the Android systems with lots of system-specific implementation details, among which *StackMap* is particularly crucial. *StackMap* is one of the most important auxiliary information to support the correct execution of binary code in ART. It represents the state mapping relationship between the physical machine and the abstract DEX virtual machine, which is required by ART for runtime support, such as stack backtracing, garbage collection, and exception handling. Any binary code level optimization should ensure the consistency between the binary code and the *stackmap* by updating it correspondingly.

4 Experimental Evaluation

4.1 Experimental Setup

The experiments were mostly conducted on a Pixel 7 Android phone equipped with the recent Tensor G2 processor and 8GB of RAM, running on the base version AOSP14 with build_id UP1A.231005.007 [31], as shown in Table 3. Note that the LTBO method using a single global suffix tree requires a memory capacity larger than 8GB. As a result, it cannot be run on the Pixel 7 directly. Given this limitation, the LTBO with a single thread suffix tree is evaluated on a desktop equipped with 64 GB of RAM and an Intel Core i9-13900K CPU, running Ubuntu 22.04 LTS.

We modified ART on the AOSP, recompiled the system image, and flashed it onto the phone for the experiments. We evaluated the following methods:

- Baseline: The original AOSP version with all available code size optimization enabled;

Table 3. Experimental setup.

Parameters	Configuration
Experiment Device	Pixel 7 Android Phone
Processor	Tensor G2
RAM	8GB
Android Version	AOSP14(UP1A.231005.007)
Test Set	Commercial Android applications in OPPO App Market

- CTO: The proposed lightweight compilation-time code outlining optimization for the three ART-specific repetitive code patterns, introduced in Section 3.1;
- LTBO: The proposed link-time outlining optimization introduced in Section 3.2 and 3.3;
- PlOpti: The paralleled suffix tree optimization introduced in Section 3.4.1, to speed up the repetitive code sequence detection and elimination work. In this evaluation, six threads are enabled to run small suffix trees in parallel. Note that existing outlining studies commonly use a single-threaded suffix tree approach;
- HfOpti: The hot function filtering optimization introduced in Section 3.4.2, which aims to mitigate the runtime performance degradation due to code outlining.

To reflect the impact on real-world mobile applications, we tested with real and heavily used commercial Android applications. We selected the top six downloaded applications from the OPPO App market, including Toutiao, Taobao, Tomato Novel, Meituan, Kuaishou, and WeChat.

These Android applications were built under the speed mode. This mode conducts a full compilation of all dex methods, aiming to optimize more aggressively to generate code that runs as fast as possible, often at the expense of other factors like binary size or compilation time.

The experiment is conducted to evaluate the proposed compilation-assisted link-time code outlining method from the following four aspects:

- The code size reduction of the OAT file on disk, i.e., the size reduction of text segment. This evaluates the saving of storage resources due to the proposed methods;
- The size reduction on the memory usage, including both data and code. This evaluates the saving of memory resources due to the proposed methods;
- The building overhead due to the proposed methods;
- The runtime performance degradation of applications due to the proposed methods.

4.2 The Code Size Reduction of The OAT File on Disk

We conducted this evaluation as follows. The application is installed on the Pixel 7 phone, and then the phone is connected via *adb* to use the *pm compile* command to generate OAT files for the application. All applications are compiled using the default compilation parameters of the Android system, and the sizes of the code segments in the generated OAT files are compared.

Table 4 illustrates the code size reduction under the speed mode. Specifically, the proposed CTO method reduces the code segment size by 3.56%, given that this method only works for three ART-specific repetitive code patterns.

Combining the CTO and the LTBO methods, the code segment size is reduced by 19.19% on average. The Kuaishou application shows the most significant reduction, with a reduction of 21.08%, while the Taobao application shows the smallest reduction, at 17.78%. Overall, for the six applications, this method can save more than 400 MB of disk space on the mobile device.

4.3 The Code Size Reduction on The Memory Usage of the Runtime OAT File

The OAT files are loaded into memory when the application runs. Therefore, reducing the code size of OAT files can effectively reduce the memory usage of the application during execution. To evaluate the reduction in memory usage of OAT files, we evaluated as follows. We perform a series of operations on each application repeatedly. To minimize operational errors, we utilized the *uiautomator* library [35] to write automated test scripts, allowing the phone to automatically carry out a series of specified operations repeatedly. We run the test script 20 times, and the average memory usage of the OAT files is recorded.

Table 5 illustrates the memory usage under the speed mode. On average, for the six tested applications, there is a reduction of 2.03% for the CTO method and 6.82% for this method combined with the LTBO method, in memory usage. This result aligns with our expectations and shows that the proposed approach is promising for mobile devices with limited memory resources.

4.4 The Building Time Overhead

The proposed CTO and LTBO methods will inevitably introduce building overhead. As stated before, we evaluated the building time of applications with a single thread suffix tree on a desktop. Then we evaluated the building time of applications with the paralleled suffix tree method on the Pixel 7. We evaluated the building time under the speed mode. Table 6 shows the building time of each application, with or without the paralleled suffix tree optimization, respectively.

It shows that for the six applications, the unoptimized code outlining method with a single thread suffix tree results in a significant overhead in building time, which slows down the building work by 489.5% on average. Please note that this slowdown is overly optimistic, considering that the Intel Core i9 processor is more powerful than the one in the Pixel 7. We analyzed the breakdown of the building overhead and found that most of the overhead comes from the LTBO work. Specifically, this is because the building and searching of

Table 4. The code size reduction of the OAT file under two optimization methods. The last three rows illustrate the ratio of code size reduction relative to the baseline.

	Toutiao	Taobao	Fanqie	Meituan	Kuaishou	Wechat	AVG
Baseline	357M	225M	264M	247M	612M	388M	/
CTO+LTBO	291M	185M	213M	201M	483M	311M	/
CTO+LTBO+PlOpti	296M	187M	221M	208M	507M	329M	/
CTO+LTBO+PlOpti+HfOpti	301M	191M	224M	211M	513M	332M	/
CTO+LTBO	**18.49%**	**17.78%**	**19.32%**	**18.62%**	**21.08%**	**19.85%**	**19.19%**
CTO+LTBO+PlOpti	**17.06%**	**16.89%**	**16.29%**	**15.79%**	**17.16%**	**15.21%**	**16.40%**
CTO+LTBO+PlOpti+HfOpti	**15.69%**	**15.11%**	**15.15%**	**14.57%**	**16.18%**	**14.43%**	**15.19%**

Table 5. The memory usage reduction under the speed mode. The last two rows illustrate the memory usage reduction relative to the baseline.

	Toutiao	Taobao	Fanqie	Meituan	Kuaishou	Wechat	AVG
Baseline	317.7M	175.4M	201.2M	238.1M	329.2M	125.7M	/
CTO	314.2M	170.6M	198.0M	238.3M	319.0M	121.0M	/
CTO+LTBO	289.3M	163.4M	186.6M	222.5M	310.7M	116.4M	/
CTO	**1.10%**	**2.74%**	**1.59%**	**-0.08%**	**3.10%**	**3.74%**	**2.03%**
CTO+LTBO	**7.26%**	**6.84%**	**7.26%**	**6.55%**	**5.62%**	**7.40%**	**6.82%**

a global suffix tree work very slowly. As the number of instructions in the tested applications reaches millions, a large number of computation tasks, as well as worsening memory efficiency due to the large working set, could significantly slow down the building work.

As a comparison, with the optimization of the paralleled suffix tree method, the average building time increases only by 71%. Here we partitioned the original suffix tree into 8 small suffix trees. The paralleled suffix tree method essentially partitions a big suffix tree into small ones, following which it builds and searches the small suffix trees in parallel. The computation task is reduced since small trees shorten the building and searching time. The memory efficiency can also be improved since only a proportion of small trees reside in memory simultaneously, leading to a much smaller working set.

The paralleled outlining method may negatively impact the effectiveness of code size reduction since the code outlining is local to each small suffix tree. However, as illustrated in the second to last line of Table 4, the impact on the code size reduction is tolerable, from 19.19% to 16.40%.

The experiments demonstrated that the proposed LTBO method, optimized with the parallel suffix tree approach, significantly mitigates the building overhead while at a tolerable loss of code size reduction, making it feasible to be utilized in a production environment. Moreover, the tradeoffs between building time and the code size reduction can be selected by adjusting the number of paralleled suffix trees, depending on the specific requirements.

4.5 The Runtime Performance Degradation

The code outlining method potentially impacts the runtime performance, since it always introduces additional call instructions. Due to the relatively small impact of the outlining method on runtime performance, the coarse-grained testing metrics, such as user experience and video frame rates, are not suitable to measure the runtime performance variation.

Moreover, to avoid testing errors caused by the phone overheating and frequency throttling, we used the CPU cycle count instead of the execution time to measure the runtime performance variation. We performed a series of operations on each application repeatedly. To minimize operational errors, we utilized the *uiautomator* library to write automated test scripts, allowing the phone to automatically carry out the specified operations repeatedly. We ran the test script 20 times, and the average CPU cycle counts were recorded.

Table 7 shows the CPU cycle counts under the speed mode. Without hot function filtering optimization, the six applications experienced an average performance degradation of 1.51%. With the hot function filtering optimization, the performance degradation is mitigated, from 1.51% to 0.90%. The applications Toutiao, Fanqie, Kuaishou and WeChat showed performance degradation below 1.0%.

The hot function filtering method may negatively impact the effectiveness of code size reduction since the hot code is excluded from the code outlining optimization. However, as illustrated in the last row of Table 4, the impact on the code size reduction is tolerable, from 16.40% to 15.19%.

Table 6. The building time under the speed mode. The last two rows illustrate the building time growth relative to the baseline.

	Toutiao	Taobao	Fanqie	Meituan	Kuaishou	Wechat	AVG
Baseline	32s	14s	23s	17s	1m01s	33s	/
CTO+LTBO	3m13s	1m31s	2m09s	1m37s	6m01s	3m05s	/
CTO+LTBO+PlOpti	55s	24s	39s	29s	1m47s	56s	/
CTO+LTBO	**503%**	**550%**	**461%**	**471%**	**492%**	**460%**	**489.5%**
CTO+LTBO+PlOpti	**71%**	**71%**	**69%**	**70%**	**75%**	**69%**	**70.8%**

Table 7. The runtime performance under the speed mode (in CPU cycle count). The last two rows illustrate the ratio of performance degradation relative to the baseline.

	Toutiao	Taobao	Fanqie	Meituan	Kuaishou	Wechat	AVG
Baseline	42107M	39274M	37216M	31270M	49734M	33041M	/
CTO+LTBO+PlOpti	42989M	39987M	37809M	31967M	50174M	33183M	/
CTO+LTBO+PlOpti+HfOpti	42384M	39798M	37524M	31931M	49939M	33031M	/
CTO+LTBO+PlOpti	**2.09%**	**1.82%**	**1.59%**	**2.23%**	**0.88%**	**0.43%**	**1.51%**
CTO+LTBO+PlOpti+HfOpti	**0.66%**	**1.33%**	**0.83%**	**2.11%**	**0.41%**	**0.03%**	**0.90%**

The experiments demonstrated that the proposed CTO and LTBO methods, optimized with hot function filtering method, could mitigates the runtime performance degradation while at a tolerable loss of code size reduction.

5 Related Work

Most related research on code size optimization is conducted during the compilation time or linking time. Traditional compilation optimization techniques, such as dead code and unreachable code elimination [8] and copy propagation [2] help reduce code size. Recent research includes code outlining [6, 11, 18, 19, 21, 25], function merging [13, 17, 27–29, 33, 34], and machine learning [4, 14, 15, 20, 24, 30, 37]. These methods have largely been effective in the mobile domain.

Function Merging. Function merging reduces code size by combining multiple identical or similar functions into one, sharing common parts of the functions while reserving the different parts with branches. This work relies mainly on fingerprint-based redundancy detection, which uses function fingerprints (typically composed of instruction opcodes and their frequencies [28]) to determine the similarity between functions. Then, sequence alignment techniques are used to determine the common parts between two functions.

The early function merging work was used to merge functions with identical parameters, return values, and function code [17, 34]. Then an extension of merging to structurally similar but not entirely identical functions was introduced[13]. Building on this, [28] proposed a sequence alignment-based function merging method capable of theoretically merging any two functions. Subsequent research [29] has primarily focused on improving this approach. Additionally, [33] introduced a novel method for identifying similar functions, significantly reducing the execution time and peak memory usage of sequence alignment-based merging techniques. However, function merging techniques will incur performance degradation and compilation overhead, and hinder the debugging process by increasing complexity, obscuring context, and complicating the analysis of errors. In addition, it is not suitable for binary code level due to correctness issues of thorough binary rewriting.

Machine Learning Based Methods. In recent years, machine learning and neural network methods have also been applied to code size reduction. Studies [4, 9, 14, 15, 20, 24, 30, 37] have introduced these approaches, which aim to select optimal optimization combinations for applications, resulting in smaller binary file sizes. There is also work to reduce code size through recurrent convolution [26]. In methodology, these methods are orthogonal to the proposed outlining method. In practice, these machine learning based methods are not suitable to reduce code size for Android applications due to their large amount of compilation time.

Other Methods. It is also found that the function inlining method may not necessarily increase code size. Instead, function inlining may reduce code size if applied carefully [10]. Some work eliminates unreachable code by analyzing runtime data of the program [1]. A new dynamic linking framework is introduced to expand the scope of code size optimization, leading to further size reduction [5].

Code Size Reduction in Android. The compiler tool of Android systems is the *dex2oat* tool, which provides optimizations of code size reduction on the HGraph phase [12]. These optimizations include dead code and unreachable code elimination, strength reduction, write barrier elimination, implicit safety point checks, loop-invariant code motion,

constant propagation [2], copy propagation, common subexpression elimination [7], partial redundancy elimination [16], and return merging [12].

6 Conclusion

We analyzed the code redundancy in the binary code of OAT files, and found several important observations. Based on these observations, we proposed a compilation-assisted link-time binary code outlining method. Experimental results show that the proposed approach reduces code size in Android applications by more than 15.19% on average, with negligible runtime performance degradation and tolerable building time overhead. Hence the proposed code outlining approach is promising for production deployment.

Acknowledgments

We thank all the reviewers for their insightful comments. This work was supported by the National Key Research and Development Program of China (No. 2022YFB4400704), National Natural Science Foundation of China (No. 62472330, No. U20A20177), the State Key Laboratory of Computer Architecture (ICT, CAS) under Grant No. CARCH A202112, Wuhan Science and Technology Joint Project for Building a Strong Transportation Country(No.2023-2-7) and OPPO Research Fund.

References

[1] Aatira Anum Ahmad, Abdul Rafae Noor, Hashim Sharif, Usama Hameed, Shoaib Asif, Mubashir Anwar, Ashish Gehani, Fareed Zaffar, and Junaid Haroon Siddiqui. 2022. Trimmer: An Automated System for Configuration-Based Software Debloating. *IEEE Transactions on Software Engineering* (2022), 3485–3505. https://doi.org/10.1109/TSE.2021.3095716

[2] V Aho Alfred, S Lam Monica, and D Ullman Jeffrey. 2007. *Compilers principles, techniques & tools.* pearson Education.

[3] Android. 2024. Simpleperf. https://developer.android.google.cn/ndk/guides/simpleperf. Accessed on: September 10, 2024.

[4] Amir Hossein Ashouri, Giovanni Mariani, Gianluca Palermo, Eunjung Park, John Cavazos, and Cristina Silvano. 2016. COBAYN: Compiler Autotuning Framework Using Bayesian Networks. *ACM Trans. Archit. Code Optim.* (2016). https://doi.org/10.1145/2928270

[5] Sean Bartell, Will Dietz, and Vikram S. Adve. 2020. Guided linking: dynamic linking without the costs. *Proc. ACM Program. Lang.* (2020). https://doi.org/10.1145/3428213

[6] Milind Chabbi, Jin Lin, and Raj Barik. 2021. An Experience with Code-Size Optimization for Production iOS Mobile Applications. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 363–377. https://doi.org/10.1109/CGO51591.2021.9370306

[7] John Cocke. 1970. Global common subexpression elimination. *SIGPLAN Not.* (1970), 20–24. https://doi.org/10.1145/390013.808480

[8] K Cooper. 2011. Torczon, L.: Engineering a Compiler.

[9] Anderson Faustino da Silva, Bruno Conde Kind, José Wesley de Souza Magalhães, Jerônimo Nunes Rocha, Breno Campos Ferreira Guimarães, and Fernando Magno Quinão Pereira. 2021. ANGHABENCH: A Suite with One Million Compilable C Benchmarks for Code-Size Reduction. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 378–390. https://doi.org/10.1109/CGO51591.2021.9370322

[10] Thaís Damásio, Vinícius Pacheco, Fabrício Goes, Fernando Pereira, and Rodrigo Rocha. 2021. Inlining for Code Size Reduction. In *Proceedings of the 25th Brazilian Symposium on Programming Languages (SBLP '21)*. 17–24. https://doi.org/10.1145/3475061.3475081

[11] Saumya K. Debray, William Evans, Robert Muth, and Bjorn De Sutter. 2000. Compiler techniques for code compaction. *ACM Trans. Program. Lang. Syst.* (2000), 378–415. https://doi.org/10.1145/349214.349233

[12] Android Developers. 2023. The Secret to Android's Improved Memory: Latest Android Runtime Update. https://android-developers.googleblog.com/2023/11/the-secret-to-androids-improved-memory-latest-android-runtime-update.html EB/OL.

[13] Tobias J.K. Edler von Koch, Björn Franke, Pranav Bhandarkar, and Anshuman Dasgupta. 2014. Exploiting function similarity for code size reduction. *SIGPLAN Not.* (2014), 85–94. https://doi.org/10.1145/2666357.2597811

[14] Anderson Faustino, Edson Borin, Fernando Pereira, Otávio Nápoli, and Vanderson Rosário. 2021. New Optimization Sequences for Code-Size Reduction for the LLVM Compilation Infrastructure. In *Proceedings of the 25th Brazilian Symposium on Programming Languages (SBLP '21)*. Association for Computing Machinery, New York, NY, USA, 33–40. https://doi.org/10.1145/3475061.3475085

[15] He Jiang, Guojun Gao, Zhilei Ren, Xin Chen, and Zhide Zhou. 2022. SMARTEST: A Surrogate-Assisted Memetic Algorithm for Code Size Reduction. *IEEE Transactions on Reliability* (2022), 190–203. https://doi.org/10.1109/TR.2021.3073960

[16] Robert Kennedy, Sun Chan, Shin-Ming Liu, Raymond Lo, Peng Tu, and Fred Chow. 1999. Partial redundancy elimination in SSA form. *ACM Trans. Program. Lang. Syst.* (1999), 627–676. https://doi.org/10.1145/319301.319348

[17] Doug Kwan, Jing Yu, and Bhaskar Janakiraman. 2012. Google's C/C++ toolchain for smart handheld devices. In *Proceedings of Technical Program of 2012 VLSI Technology, System and Application*. 1–4. https://doi.org/10.1109/VLSI-TSA.2012.6210142

[18] Kyungwoo Lee, Ellis Hoag, and Nikolai Tillmann. 2022. Efficient profile-guided size optimization for native mobile applications. In *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction (CC 2022)*. Association for Computing Machinery, New York, NY, USA, 243–253. https://doi.org/10.1145/3497776.3517764

[19] Gai Liu, Umar Farooq, Chengyan Zhao, Xia Liu, and Nian Sun. 2023. Linker Code Size Optimization for Native Mobile Applications. In *Proceedings of the 32nd ACM SIGPLAN International Conference on Compiler Construction (CC 2023)*. Association for Computing Machinery, New York, NY, USA, 168–179. https://doi.org/10.1145/3578360.3580256

[20] Raphael Mosaner, David Leopoldseder, Lukas Stadler, and Hanspeter Mössenböck. 2021. Using machine learning to predict the code size impact of duplication heuristics in a dynamic compiler. In *Proceedings of the 18th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes (MPLR 2021)*. 127–135. https://doi.org/10.1145/3475738.3480943

[21] Jessica Paquette. 2016. Reducing code size using outlining. https://llvm.org/devmtg/2016-11/Slides/Paquette-Outliner.pdf. Accessed on: September 10, 2024.

[22] Google Play. 2024. App Bundle on Google Play. https://developer.android.com/google/play/expansion-files. Accessed on: October 22, 2024.

[23] Google Play. 2024. Download size restriction on Google Play. https://developer.android.com/topic/performance/reduce-apk-size. Accessed on: October 22, 2024.

[24] Suresh Purini and Lakshya Jain. 2013. Finding good optimization sequences covering program space. *ACM Trans. Archit. Code Optim.* (2013). https://doi.org/10.1145/2400682.2400715

[25] River Riddle. 2017. Interprocedural IR Outlining For Code Size. https://llvm.org/devmtg/2017-10/slides/Riddle-InterproceduralIROutliningForCodeSize.pdf. Accessed on:

September 10, 2024.

[26] Rodrigo C. O. Rocha, Pavlos Petoumenos, Björn Franke, Pramod Bhatotia, and Michael O'Boyle. 2022. Loop Rolling for Code Size Reduction. In *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 217–229. https://doi.org/10.1109/CGO53902.2022.9741256

[27] Rodrigo C. O. Rocha, Pavlos Petoumenos, Zheng Wang, Murray Cole, Kim Hazelwood, and Hugh Leather. 2021. HyFM: function merging for free. In *Proceedings of the 22nd ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES 2021)*. 110–121. https://doi.org/10.1145/3461648.3463852

[28] Rodrigo C. O. Rocha, Pavlos Petoumenos, Zheng Wang, Murray Cole, and Hugh Leather. 2019. Function Merging by Sequence Alignment. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 149–163. https://doi.org/10.1109/CGO.2019.8661174

[29] Rodrigo C. O. Rocha, Pavlos Petoumenos, Zheng Wang, Murray Cole, and Hugh Leather. 2020. Effective function merging in the SSA form. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 854–868. https://doi.org/10.1145/3385412.3386030

[30] Anderson Faustino da Silva, Bernardo N. B. de Lima, and Fernando Magno Quintão Pereira. 2021. Exploring the space of optimization sequences for code-size reduction: insights and tools. In *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction (CC 2021)*. Association for Computing Machinery, New York, NY, USA, 47–58. https://doi.org/10.1145/3446804.3446849

[31] Android source code. [n. d.]. The Secret to Android's Improved Memory: Latest Android Runtime Update. https://cs.android.com/android/platform/superproject/main EB/OL.

[32] Statcounter. 2024. Android market share. https://gs.statcounter.com/os-market-share/mobile/worldwide. Accessed on: October 22, 2024.

[33] Sean Stirling, Rocha Rodrigo C. O., Kim Hazelwood, Hugh Leather, Michael O'Boyle, and Pavlos Petoumenos. 2022. F3M: Fast Focused Function Merging. In *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 242–253. https://doi.org/10.1109/CGO53902.2022.9741269

[34] Sriraman Tallam, Cary Coutant, Ian L Taylor, David X Li, and Chris Demetriou. 2010. Safe ICF: Pointer safe and unwinding aware identical code folding in the gold linker. *Proceedings of the 2010 GCC Summit* (2010), 107–114.

[35] The uiautomator testing framework. 2024. The uiautomator testing framework. https://emanual.github.io/Android-docs/tools/help/uiautomator/index.html. Accessed on: September 10, 2024.

[36] E. Ukkonen. 1995. On-line construction of suffix trees. *Algorithmica* (1995), 249–260.

[37] André Felipe Zanella, Anderson Faustino da Silva, and Fernando Magno Quintão. 2020. YACOS: a Complete Infrastructure to the Design and Exploration of Code Optimization Sequences. In *Proceedings of the 24th Brazilian Symposium on Context-Oriented Programming and Advanced Modularity (SBLP '20)*. 56–63. https://doi.org/10.1145/3427081.3427089

Received 2024-09-12; accepted 2024-11-04