Linker Code Size Optimization for Native Mobile Applications

Gai Liu
ByteDance
Mountain View, CA, USA
gai.liu@bytedance.com

Umar Farooq
ByteDance
Mountain View, CA, USA
umarfarooq@bytedance.com

Chengyan Zhao
ByteDance
Mountain View, CA, USA
chengyan.zhao@bytedance.com

Xia Liu
ByteDance
Shenzhen, China
liuxia.nathan@bytedance.com

Nian Sun
ByteDance
Shanghai, China
sunnian@bytedance.com

Abstract

Modern mobile applications have grown rapidly in binary size, which restricts user growth and hinders updates for existing users. Thus, reducing the binary size is important for application developers. Recent studies have shown the possibility of using link-time code size optimizations by re-invoking certain compiler optimizations on the linked intermediate representation of the program. However, such methods often incur significant build time overhead and require intrusive changes to the existing build pipeline.

In this paper, we propose several novel optimization techniques that do not require significant customization to the build pipeline and reduce binary size with low build time overhead. As opposed to re-invoking the compiler during link time, we perform true linker optimization directly as optimization passes within the linker. This enables more optimization opportunities such as pre-compiled libraries that prior work often could not optimize. We evaluate our techniques on several commercial iOS applications including NewsFeedApp, ShortVideoApp, and CollaborationSuiteApp, each with hundreds of millions of daily active users. Our techniques on average achieve 18.4% binary size reduction across the three commercial applications without any user-perceivable performance degradations.

CCS Concepts: • **Software and its engineering → Compilers**; • **Human-centered computing → Ubiquitous and mobile computing**.

CC '23, February 25–26, 2023, Montréal, QC, Canada

ACM ISBN 979-8-4007-0088-0/23/02…$15.00
https://doi.org/10.1145/3578360.3580256

Keywords: Code Size Optimization, Static Analysis, iOS

ACM Reference Format:
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 '23), February 25–26, 2023, Montréal, QC, Canada.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3578360.3580256

1 Introduction

Mobile applications have seen tremendous adoption over the last decade. Today, billions of users depend on them for a variety of reasons, including access to news, social media, ride sharing, work productivity, and much more. In this competitive and vastly growing environment, constantly delivering new features is of prime importance to application developers. However, the proliferation of new features results in a huge increase in binary size [1, 2]. At the same time, mobile devices provide limited storage, and distribution channels (i.e., app stores) enforce download-size restrictions. For example, the Apple App Store [3] requires a Wi-Fi connection to download applications larger than 200 MB as of 2020. This download-size restriction by the app store limits application growth as new installations and updates, including security improvement, cannot be performed without a Wi-Fi connection.

Code size optimization. Compiler optimizations are effective in minimizing the size of the compiled binary [4]. In addition to performance benefits, many compiler optimizations may also reduce code size, such as dead and unreachable code elimination [5], common sub-expression elimination [6], partial redundancy elimination [7], constant and copy propagation [8, 9], constant folding [10], value numbering [11], register allocation and instruction scheduling [12], code compression [13–15], and peephole optimizations [16]. Other than compiler optimizations, link-time optimizations (LTO) and post-link-time optimizations [17–26] have also shown success in reducing code size.

Table 1. Summary of representative size optimization approaches for native mobile applications. Build time overhead is the increase in build time when enabling the corresponding techniques.

Approach	Build time overhead	Require custom build pipeline	Binary size saving
Chabbi et al. [29]	200%[1]	Yes	17.6%
Lee et al. [30]	40%	Yes	12.6%
This work	17%	No	18.4%

State of the art. iOS applications are commonly compiled using LLVM [27], and several of the above-mentioned optimizations are available in LLVM by default. One of the key size optimization passes is the machine outlining pass that extracts frequent sequences of instructions into separate functions at the machine intermediate representation (IR) level to reduce the code size [28]. The machine outlining pass is scheduled as part of the compilation pipeline, which operates on a single compilation unit during compilation. Optimizing only within a single compilation unit leaves much room for size reduction on large applications. This is because a significant portion of the repetitive patterns is common across multiple files/compilation units.

To enable global size optimizations across compilation units, state-of-the-art approaches often utilize recent development in LTO [17, 31]. Chabbi et al. [29] proposed a whole-program machine code outlining using full LTO (also known as monolithic LTO). The full LTO process merges all the LLVM IR files generated from the input files into a single IR module, and applies transformation and lowering passes on the merged module. While this approach provides a significant binary size reduction, it contributes up to 45 minutes overhead in the build pipeline since the merged module can only be processed by a single thread. This overhead triples the overall build time [29]. Additionally, this approach requires the source files or the IRs to be available for optimization[2], which means binary-only third-party libraries cannot be effectively optimized. On top of this, a customized build pipeline for linking the IRs needs to be built, which incurs non-trivial changes to the existing flow [29].

More recently, Lee et al. [30] extended the machine outlining pass and modifies the LLVM compilation pipeline to enable global machine outlining. The authors proposed to run the code generation step twice. In the first iteration, they traverse each file to collect optimization opportunities. In the second round, actual optimizations are performed. Their approach requires significant changes to the compilation pipeline and the two-round code generation incurs around 40% build time overhead. Since their approach works at the IR level, source files would be needed to be optimized. Collectively, these efforts introduce effective code size optimizations for iOS applications. Nonetheless, eliminating the requirement of a custom build pipeline, reducing the build time overhead, and extending the optimizations to third-party libraries remain the challenges to solve. We compare our approach and other existing techniques in Table 1, where the binary size saving numbers are taken from the apps with the largest size reductions in [29] and [30], respectively.

Overview of this work. To address these challenges, we propose a novel framework to perform linker optimization[3] to reduce the binary size, which does not require customizing the build pipeline and the time overhead remains within 17% of the overall build time. We extend the open-source `ld64` linker [33] with additional analyses and size-optimizing transformation passes. This enables the build pipeline to leverage the optimizations by simply using our customized linker without needing to change any existing compiler/linker flags. Specifically, our instruction decoding utilities allow the linker to understand the semantics of the machine instructions. Our extension to function hashing enables efficient identity check. The instruction visibility analysis ensures safe code transformations that maintain the correct semantics of the program and its associated metadata. These analyses enable us to perform code size optimizations, including general sequence outlining, frame code outlining, and identical code folding. We evaluate our linker on three widely used commercial mobile applications in terms of code size reduction and build time overhead. Our evaluations show that our technique reduces the binary size of NewsFeedApp, ShortVideoApp, and CollaborationSuiteApp by 17.8%, 20.5%, and 17.1%, respectively.

In this work, we make the following contributions:

- To the best of our knowledge, we are the first to propose conducting code size optimization within the linker, as opposed to the existing approach of piggybacking on the compiler's optimization passes.
- We describe a novel framework within the iOS linker for code size optimization including the necessary analyses and code transformations.
- We show that our techniques achieve best-in-class results in both code size and build time on several real-world iOS applications without user-noticeable performance degradations.

Next, we describe our techniques, including analyses and optimizations in Section 2. We discuss implementation details in Section 3, then we present evaluations and experiments in Section 4, followed by the related work in Section 5. Finally, we conclude this work in Section 6.

[1] This technique triples the build time according to [29].

[2] Embedding LLVM IR in the object files requires explicitly specifying `-fembed-bitcode` option, which is off by default in Clang.

[3] In this work, link time optimization denotes the existing approach of invoking the optimizer during linking through a shared object such as `libLTO` [32]. Linker optimization refers to our approach of directly implementing optimization passes in the linker.

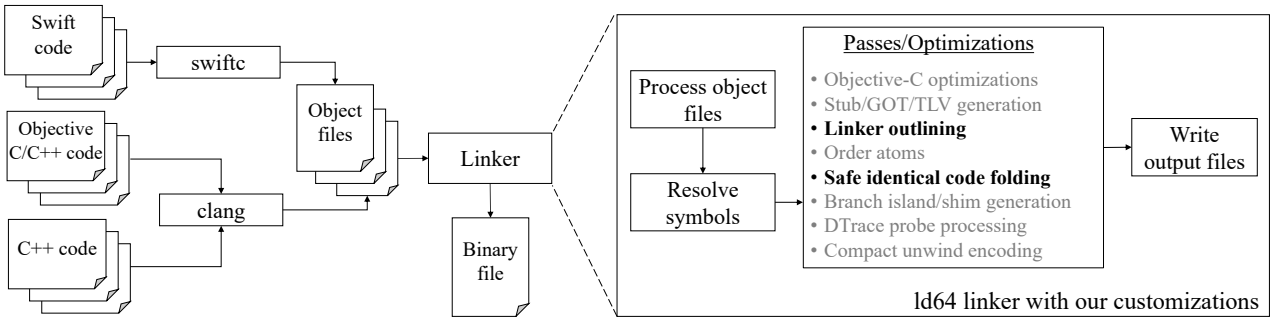


Figure 1. Overview of the build pipeline and detailed view of the `ld64` linker with our new passes highlighted.

2 Techniques

In this section, we describe the main techniques to reduce the code size of native iOS applications. We first discuss the common analyses that we use across the optimizations, then present the specific optimizations that effectively reduce the code size. We propose two main types of code size reduction techniques, (i) linker outlining aims at finding repetitive sequences of instructions within functions and then outlining them into a shared function, and (ii) safe identical code folding explores repetitive functions and merges them into a single function while ensuring the correctness under function pointer comparisons. Figure 1 shows the general build pipeline, and we highlight our contributions to the linker in the flow.

2.1 Analyses

To enable code transformations during linking, we develop a few useful analyses on the machine instruction level. This enables us to analyze the function-level and instruction-level properties and make optimization decisions.

Instruction decoding utilities. Traditional linkers do not have the full instruction decoding capability since they do not need to know the fine details of all instructions. However, in our work, it is critical that we fully decode the instructions and unveil details including opcode, immediate values, register usage, and any special mode flags. We construct a comprehensive set of utility functions in the `ld64` linker for the AArch64 ISA [34] and use them throughout the analyses and optimizations. The utility functions resemble functionalities that are commonly found in a typical compiler where information such as the type, opcode, and register indices of an instruction is obtained from its binary encoding.

Function hashing. Hashing a function is a common routine used in our optimizations. It enables efficient identity checks across multiple functions, which is an important operation in code folding. Here, we extend the existing function hashing utility in `ld64`. In our scheme, we hash each function into a 64-bit unsigned integer. Specifically, the hash of a function is determined by two factors. First, the machine instructions

```
104a5f9fc:  mov x20, x0
104a5fa00: ldr x0, [x19, #72]
104a5fa04: cbz x0, 0x104a5fa18
104a5fa08: ldr x22, [x0]
104a5fa0c: bl 0x10c483fd0
104a5fa10: mov x0, x22
104a5fa14: cbnz x22, 0x104a5fa08
104a5fa18: ldr x0, [x21]
```

Figure 2. An example of instruction visibility where the two highlighted `ldr` instructions are the branch targets of the corresponding `cbz`/`cbnz` instructions. They are marked as visible.

are iteratively hashed with a prime multiplier. Second, the metadata (fixups in the `ld64` context) is hashed from strings to integers and added to the hash value from the first step. This two-step hashing scheme ensures control flow information such as branch targets are encoded in the computed hash.

The necessary condition for two functions to be identical is that their hash values are identical. However, having identical hash values is not sufficient to prove that two functions are identical due to potential hash collisions. Bytewise comparison is needed to prove sufficiency. To this end, we implemented an optional safety check in the pass to conduct byte-wise identity check across all functions with the same hash value. Since the number of functions mapping to the same hash value is relatively small on average compared to the total number of functions in the program, the compile time overhead of this safety check is small.

Instruction visibility. Since our optimizations are conducted during the late stage of the build pipeline, we need to be aware not to optimize away certain instructions that other instructions or metadata may explicitly reference. We define *visible* instructions to be the branch targets of control flow instructions and the start/end/jump targets in an exception handling table. Removing such visible instructions could lead to unexpected application behaviors such as incorrect logic and inaccurate exception handling. To understand their

semantics and identify these special instructions, we linearly scan the instruction sequence and parse the exception handling tables embedded in the object files. Information related to the visible instructions is preserved upon discovery and reused by various optimization passes during linking. Figure 2 shows an example of visible instructions. The two highlighted `ldr` instructions are the branch targets of their corresponding `cbz`/`cbnz` instructions. Removing or outlining these instructions may lead to incorrect behaviors, thus we consider them visible to other instructions and skip them during outlining.

2.2 General Sequence Outlining

Outlining is a key technique in reducing the code size. Outlining discovers common code sequences in a program and replaces them with calls to the corresponding outlined sequences. Traditionally, outlining is commonly done during compilation. For example, LLVM employs a machine-outliner pass that operates on the machine IR. When combined with full LTO, LLVM-based outlining can explore optimization opportunities at the whole program granularity [29]. We instead propose and implement outlining inside a linker for the following reasons. First, mobile applications are usually written in a modularized fashion, and it is common to include third-party libraries which are pre-compiled into object code. This development flow is not compatible with the LLVM-based outlining solution since the source code of many modules is not available during build time. Second, full LTO based whole program outlining significantly increases the build time by tens of minutes to a few hours [29]. Such a long compile time poses a significant challenge for integration into rapid development pipelines. Third, full LTO based build flow complicates incremental compilation and makes incremental debugging significantly more difficult.

We develop a general code sequence outliner in the `ld64` linker that can optimize the whole program. For example, our linker outliner can optimize third-party libraries available only in binary format, which is beyond the capability of the LLVM machine outliner. Algorithm 1 details the steps in our general sequence outliner. Our outliner first traverses the entire program and hashes every instruction sequence whose length is within a predefined range (e.g., from length-2 to length-12). We hash the instruction sequences by extending the function hashing technique introduced in Section 2.1 to handle arbitrary code sequences. The range of sequence lengths is a user-configurable parameter, and we empirically observe that sequences longer than 12 instructions are rarely repeated in our applications. During the traversal, we keep track of the length of each hashed sequence, and their occurrences efficiently by extensive hashing and caching. We then employ a cost function to evaluate the profitability of a given sequence, where both the length and the occurrences matter. We provide linker flags to control the cost function's aggressiveness. Next, we create the outlined functions and

```
mov x21, x0          ldp d2, d3, [x8, #16]    mov w3, #0
mov x0, x19          b <objc_msgSend>         b <_isPlatformVersionAtLeast>
mov x1, x20
```

(a) **(b)** **(c)**

Figure 3. Examples of highly repeated sequences as good outlining candidates: (a) data movements between registers; (b) calling Objective-C runtime function; (c) calling system function.

modify the control flow of the original code to branch to these outlined functions. An outlined function either inherits the control flow from its original sequence or returns control back to the caller once the outlined function finishes its execution. Finally, we update the relevant metadata to reflect the changes due to outlining. This includes updates to the exception handling table and the debug information. We choose to use this linear scan based algorithm mainly due to its simplicity to implement and debug, its linear time complexity, and that it exhaustively covers all instruction sequences of selected lengths.

Figure 3 shows three examples of instruction sequences that are outlined in one of our applications, including a sequence of data movements between registers, calling an Objective-C runtime function, and calling a system function. Each of the three sequences appeared more than 500 times in the application.

Update branch targets. Since outlining modifies the semantics and positions of instructions, we need to update the relevant control flow instructions for correctness. This includes both direct and indirect branches.

For direct branch instructions where the branch offset is hardcoded in the instruction's encoding, we first identify the target instruction before outlining by decoding the offset value. During transformation, we record the mapping of the instruction indices before and after outlining, so that we can update the branch offset values to point to the correct targets when we write out the instruction sequences after outlining.

For indirect branches where the targets are encoded as data-in-code (e.g., jump tables), we currently skip the outlining optimization altogether on the particular function. Modifying the function without updating the content of data-in-code would lead to incorrect logic. In `ld64` terms, we identify such functions by looking for `kindDataInCode` type of fixups in an atom. We empirically observe that less than 2% of the overall functions in our applications contain data-in-code components, thus skipping such functions has little impact on the overall size reduction. Alternatively, one could parse these data-in-code components and update their contents based on changes made by the outliner.

For indirect branches where the targets are expressed as linker relocations, since our outlining pass is scheduled before atom ordering and fixup resolution, such relocations

Algorithm 1: General Sequence Outlining

```
Input: Program // program to be optimized
       Length // longest sequence to consider
       MinFreq // frequency threshold
Output: optimizedProgram

// Step 1: collect potential outline sequences
foreach function ∈ Program do
    visibleSet ⟵ ComputeVisibility(function)
    for len ← 2 to Length do
        // sequence is of length len
        foreach sequence ∈ function do
            skip = False
            foreach inst ∈ sequence do
                if inst ∈ visibleSet then skip = True ;
            if skip then continue;
            h ← hash(sequence)
            // map hash value to frequency
            hashToFrequency[h]+ = 1
            // map hash value to list of its
               callsites
            hashToCallSites[h].append(sequence)
// byte-wise identity check on sequences with
   the same hash value
verifySequences(hashToFrequency, hashToCallSites)
// Step 2: make outline decisions
// sort sequences by length and frequency
sort(hashToFrequency)
foreach hash ∈ hashToFrequency do
    if hashToFrequency[hash] > MinFreq then
        outlineDecisions.append(hashToCallSites[hash])
// Step 3: create outlined functions
foreach outlineInfo ∈ outlineDecisions do
    // create new function with outlined
       sequences
    createOutlineFunc(outlineInfo[0])
// Step 4: modify original functions
// find out all functions that need updates and
   the corresponding instructions to be outlined
callSites ← collectCallSites(hashToCallSites)
 foreach CS ∈ callSites do
    foreach outlineSequence ∈ CS do
        // replace to-be-outlined sequences with
           branching logic
        createBranchingLogic(outlineSequence)
    // update branch target indices for control
       flow instructions
    updateBranchTarget(CS)
    // update metadata (i.e., exception handling
       table) of the function
    updateMetadata(CS)
```

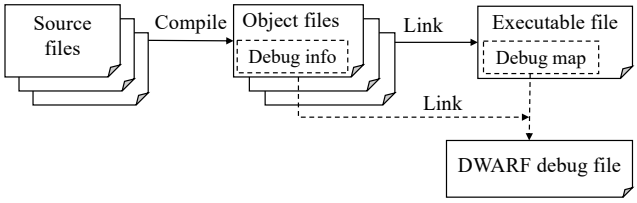


Figure 4. An illustration of the DWARF debug information linking flow.

are symbolically represented during our passes. Thus, we do not need to explicitly update such symbolic relocations. Instead, we rely on linker's downstream fixup resolution step to correctly replace the symbolic fixups with the final addresses of the corresponding functions.

Update exception handling table. The exception handling table (EHT) describes the program behavior when an exception happens, including the execution of exception handling code and any cleanup or stack unwinding actions. The EHT is encoded in a language-specific data area of the binary. In each code segment where an exception can potentially happen, the EHT describes both the landing pad location (if any) and the required actions (e.g., calling an object destructor). The EHT refers to instructions in the code segments using their relative indices within the function they reside in. As a result, when we modify the code sequences during outlining, we also need to update the content of the EHT to reflect the modification. To this end, we provide a *parser* that parses an EHT, and a *rewriter* that updates the parsed EHT based on the changes made by the outliner.

Update debug information. Debug information (e.g., in DWARF format [35]) is crucial when running a debugger and analyzing crashes. Debug information allows mapping an instruction's address to its original location in the source code. In a typical compile and link flow, such as the one for iOS applications, the pre-linking debug information is firstly generated by the compiler and stored as part of the object file. Then at link time, the linker generates a debug map that records the mapping of each function to its final address. Finally, the debug information linker (e.g., DWARF linker) utilizes the debug map to collect the debug information from each object file and link them to the final DWARF file. Figure 4 shows a high-level view of the DWARF linking process.

One important assumption of the above-mentioned DWARF linking flow is that functions in the final linked binary are identical to their counterparts in the pre-linking object files. In other words, the linker should not modify the content or the size of the functions. However, since our linker outliner modifies the functions to reduce their sizes, this assumption is no longer valid. Running the linker outlining pass without

updating the DWARF information would lead to inconsistencies between the executable and the debug information.

We customized the DWARF linker (i.e., the `dsymutil` tool) to account for the changes made by the outlining pass. To achieve this, the `ld64` linker writes a lightweight auxiliary file containing the information on how the outliner modified the content of the functions. Then the customized DWARF linker reads this file and adjusts the address mapping based on modifications made by the linker outliner. We provide a bundled executable linker with the customized DWARF linker for building the application.

2.3 Frame Code Outlining

Many calling conventions require the callee to explicitly save and restore callee-saved registers within prolog and epilog regions of a function. Such logic, also known as the frame code, is highly regular and can consume non-trivial space. Linker frame code outlining aims at extracting such code regions into shared functions and replacing the original frame code sequences with calls to these shared functions.[4] Figure 5a shows an example of a prolog frame code that first updates the stack pointer, then stores callee-saved registers (`x19` to `x30`) on the stack, and finally updates the frame pointer register `x29`.

Unlike the general sequence outlining discussed in Section 2.2, frame code sequences are more regular. We specialize our optimization to better utilize the regularity. For example, prolog (epilog) code only appears at the beginning (end) of a function. When optimizing the prolog segments, there is no need to preserve the current values of temporary registers since they do not contain live values when execution enters the current function. This is especially useful since we can use the temporary registers to store return addresses when calling the outlined prolog sequences. Similar reasoning is also applicable to optimize the epilog segments since no temporary registers will be live after the epilog sequence.

Normalize stack offset value. A compiler can reduce the number of stack offset adjustment operations by merging the offset adjustment due to callee-saved registers with those related to other temporary variables on the stack. In Figure 5a, the prolog reserves a total of 320 bytes of space in the stack for the current function, among which 96 bytes are reserved for the callee-saved registers, and the remaining 224 bytes for other temporary variables declared in the current function. In our frame code outlining pass, we add a normalization step to separate these two sources of stack offset adjustments. Figure 5b shows the normalized sequence, where we moved the stack pointer adjustment to the end of the sequence and updated the constant offset values accordingly. As a result,

```
sub sp, sp, #320
stp x28, x27, [sp, #224]
stp x26, x25, [sp, #240]
stp x24, x23, [sp, #256]
stp x22, x21, [sp, #272]
stp x20, x19, [sp, #288]
stp x29, x30, [sp, #304]
add x29, sp, #304
```

(a)

```
stp x28, x27, [sp, #-96]
stp x26, x25, [sp, #-80]
stp x24, x23, [sp, #-64]
stp x22, x21, [sp, #-48]
stp x20, x19, [sp, #-32]
stp x29, x30, [sp, #-16]
sub x29, sp, #16
sub sp, sp, #320
```

(b)

Figure 5. Frame code sequence and normalization: (a) an example sequence of prolog frame code; (b) the equivalent prolog sequence after normalizing the stack offset.

the normalized frame code sequences from different functions can share the same outlined framecode function if they store/load the same number of callee-saved registers. This normalization step reduces the number of outlined functions that need to be created, thus improving the total size saving.

In this step, we constrain the transformation so that we only write temporary variables within the stack's red zone (e.g., 128 bytes for AArch64 architecture [36]). The application binary interface ensures that these locations within the red zone will not be modified by other parts of the system. For platforms without a defined stack red zone, we skip this normalization step to ensure safety.

2.4 Safe Identical Code Folding

While outlining explores deduplication at the instruction level, identical code folding (ICF) reduces code size by discovering functions with identical implementations and replacing them with a shared implementation. The original `ld64` linker includes a code deduplication pass where only functions with the "autohide" property are considered. This limits deduplication's applicable scope because the pass relies on the compiler to explicitly mark qualified functions. We find that there are a large number of symbols (especially private symbols) that are not marked as "autohide", leaving significant room for improvement. We thus customize the deduplication pass in `ld64` by considering all non-global symbols as candidates for ICF.

Figure 6 shows an example of three identical functions that can be merged to reduce the code size. These are commonly setter and getter methods of variables, which can be either manually written or automatically synthesized in languages such as Objective-C. These three functions are from entirely independent modules, but they all implement the same logic that stores one byte of data into memory. In one of our applications, there are more than 2,000 such functions with the exact two-instruction sequence across different compilation units, making them ideal candidates for ICF.

[4] We note that a prior work [30] proposed a frame code optimization technique during LLVM machine IR optimization, while we conduct frame code outlining during linking.

```
1048965a4 <-[VideoPlayerMonitor setFromBackground:]>:
  02 20 00 39   strb   w2, [x0, #8]
  c0  03 5f d6  ret

1048c5368 <-[CrashKit setNeedEncrypt:]>:
  02 20 00 39   strb   w2, [x0, #8]
  c0  03 5f d6  ret

104d26c60 <-[AudioPlayerData setIsCommunity:]>:
  02 20 00 39   strb   w2, [x0, #8]
  c0  03 5f d6  ret
```

Figure 6. An example of identical functions that can be merged. The function names are modified for confidentiality.

Check identical functions. The ICF pass first computes a hash for every non-global function using the function hashing technique described in Section 2.1. It then groups the functions with the same hash value and checks whether folding can reduce the overall code size.

Handle function pointer comparisons. A function pointer stores the start address of a function in the memory. In many languages, such as C, C++, or Objective-C, the programmer or the runtime is allowed to conduct arithmetic operations over function pointers. One of the most used arithmetic operations is equality comparison. It checks whether two function pointers point to the same address (i.e., the function's implementation). Figure 7a shows a toy example of using function pointer comparison, where we assume that the implementations of both `func1` and `func2` are identical.

A straightforward ICF implementation (commonly known as the `icf=all` option in modern linkers) directly replaces the duplicate's implementation address with the address of the other function. Such a transformation would lead to an incorrect return value of 1 for this example. To ensure safety under function pointer comparison, we improve the `ld64` linker by implementing the `icf_safe` option.[5] The `icf_safe` option adds redirection logic to preserve correctness under function pointer equality comparison. Figure 7b illustrates the assembly code generated by `icf_safe`. Instead of directly replacing the implementation address of `func1` with `func2`, we add a single-instruction redirection logic to branch to `func2` inside `func1`, thus preserving the original behavior under function pointer equality comparison.

The safe ICF incurs an one-instruction overhead compared to the `icf=all` option. We empirically observe that the additional branching logic introduced by the `icf_safe` pass does not have any visible performance impact across the applications we tested compared to the baseline version without ICF enabled.

[5]The `icf_safe` option is available in some other linkers [37] with potentially different implementations. Our work introduces an implementation in the `ld64` linker.

```
void (*fp1)(int) = func1;
void (*fp2)(int) = func2;
// func1 and func2 are identical
int func_ptr_compare() {
  return (fp1 != fp2) ? 0 : 1;
}
```

(a) An example of function pointer comparison.

```
0x10000 <func1>:
  b 0x10004 <func2>

0x10004 <func2>:
  …
```

(b) Illustration of assembly code generated by `icf_safe`.

Figure 7. Handling function pointer comparisons in `icf_safe`.

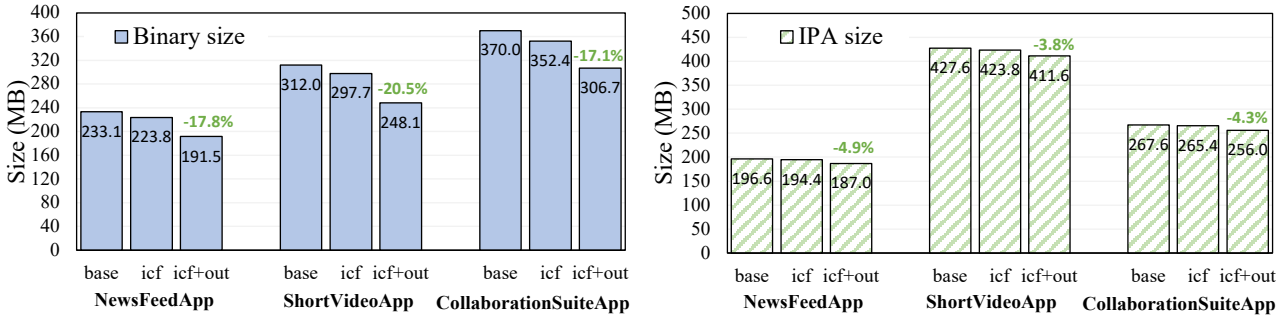
3 Implementation

We implement previously described ICF and outlining optimizations as additional passes in the Apple `ld64` linker (version 609) [33]. Our implementation consists of more than 3,500 lines of source code, written in the C++ programming language. The outline pass includes both the general sequence outlining and the frame code outlining as discussed in Section 2. We schedule these new optimization passes in the `ld64` linker after object file parsing and symbol resolution. The outlining pass is scheduled before the ordering pass that determines the total order of all the atoms. This is because the outlining pass creates new atoms. Together with those existing ones, the new atoms need to be ordered by the ordering pass. The safe ICF optimization is implemented as part of the existing code deduplication pass in `ld64`, which is scheduled right after the ordering pass. The overall `ld64` linking flow and optimization pipeline are shown in Figure 1. By default, the optimizations target AArch64 architecture [34] since it is the dominating architecture for iOS production devices. The general optimization principles are equally applicable to other CPU architectures as well.

4 Experiments

To evaluate the effectiveness and performance of our optimization passes, we conduct extensive experiments on iOS applications.

Applications. We apply our optimizations to three widely used commercial iOS applications. Each of these iOS applications has hundreds of millions of daily active users and covers a wide range of mobile application usage scenarios. NewsFeedApp is a news recommendation application, which provides personalized text, audio, and video content to end users. ShortVideoApp is a mobile short video hosting and sharing application. It hosts a variety of user-created short videos lasting between tens of seconds and a few minutes. It also personalizes users' video feed using machine learning based recommendations. Lastly, CollaborationSuiteApp is an iOS client of an enterprise collaboration platform with services covering email, instant messaging, video/audio conferencing, audio transcription, online documentation, and



(a) Binary size: size of the uncompressed binary executable. **(b)** IPA size: download size of the compressed application package.

Figure 8. Size optimization for NewsFeedApp, ShortVideoApp and CollaborationSuiteApp. `base`: baseline approach with no linker size optimizations. `icf`: enable the safe ICF optimization. `icf+out`: enable both safe ICF and linker outlining.

so on. All these applications are written using a mixture of Objective-C, Swift, and C++. One of the applications contains components written in Rust. The number of functions in each application ranges from 1×10^6 to 2×10^6 functions.

Experimental setup. To build the above-mentioned commercial applications, we perform build on an idle build machine with a 2.6 GHz 6-core Intel CPU and 64 GB of RAM running MacOS. The applications are compiled with Apple's Xcode toolchain (version 13.0) [38] and linked using the `ld64` linker with our custom passes.

Next, we discuss our experiment details in terms of code size reduction and build time comparison. Furthermore, we compare performance impacts with and without our optimizations.

4.1 Code Size Reduction

Figure 8 shows the size reduction for each application when enabling the two optimization passes. The compiler flags in the baseline have been tuned manually on a per-module basis to satisfy the module's specific quality goals independent of our work. For example, video editing/playing related modules are compiled with `-O3` for maximum performance, while other less computationally intensive modules are compiled with `-Oz` or `-Os` to minimize code size. In this experiment, we do not change any existing compiler flag that the build system is already using. Instead, we only add the `icf_safe` and `outline` linker flags. This helps to ensure a fair comparison.

The size reduction is measured in two ways. Binary size measures the size of a binary file produced by a linker. It can be either a library or an executable. A binary file is commonly composed of multiple sections including both text section(s) and data section(s). Our optimization passes only operate on the text section(s) inside a binary file. The size of an iOS app store package (i.e., IPA) refers to the size of an application package delivered through the Apple App Store [3]. It contains both binary files and resource files, including images, language support files, resource bundles, and so on. The IPA file is usually downloaded in a compressed format and then decompressed during installation.

Across all three applications, the two optimizations combined achieved 18.4% size reduction in uncompressed binary, and 4.3% size reduction in the IPA file on average. Specifically, the ICF pass reduces the size of the uncompressed applications by 4.4% and the linker outlining pass further reduces the uncompressed size by 14.0% on average. We observe that our approach achieves more significant size savings in both the binary size and the IPA size than start-of-the-art LTO based approaches [29, 30]. In addition, our results agree with [30] in that the reduction in compressed IPA size is much smaller than that in binary size. This is likely because the IPA compressor is more effective on binaries than other resources.

4.2 Build Time Comparisons

In our build configuration, the two size optimization passes are enabled only in release build, and debug build is entirely intact. To measure the link time impact, we profile the link time increase due to the two passes.

Table 2 shows the build time breakdown for the three applications and link time consumed by the two passes individually. The total build time is the time taken from setting up the build environment to the linker finished producing the final IPA file. The build process includes cloning source repositories, compiling source files individually to produce object files, and linking them together to produce the final images. We observe that the `icf_safe` pass increases the link time by 2.0% on average, while the `outline` pass accounts for 14.8% of the link time across the three applications. We observe that the build time is vastly dominated by time spent compiling source files. Overall, the two passes collectively incur a 16.7% increase in the overall build time. These two passes are enabled only in the release stage and do not impact application developers' feature development workflow. The increase in build time is mild and considered acceptable by the applications' build teams.

Table 2. Build time profile on NewsFeedApp, ShortVideoApp and CollaborationSuiteApp and the percentage overhead of our passes. m: minutes, s: seconds.

	NewsFeedApp	ShortVideoApp	CollaborationSuiteApp	Geomean
Setup and compile	39m8s	26m47s	18m57s	–
Link	13m23s	22m42s	9m12s	–
- `icf_safe` pass	45s	1m35s	22s	–
- `outline` pass	6m15s	8m21s	4m24s	–
Total build time	52m31s	49m29s	28m9s	–
`icf_safe` build time overhead	1.4%	3.2%	1.3%	2.0%
`outline` build time overhead	11.9%	16.9%	15.6%	14.8%
`icf_safe` + `outline` build time overhead	13.3%	20.1%	16.9%	16.7%

Table 3. Application startup time comparison with NewsFeedApp. Baseline: results with default flags; Size Optimized: results with `icf_safe` and `outline` on. The numbers are averaged over five runs.

	Baseline (ms)	Size Optimized (ms)
Library loading	269.0	254.5
Object loading	205.6	213.7
Application Launching	240.6	241.8
Initial Frame Rendering	107.0	112.9
Total	822.2	822.9

Table 4. Video playing frames-per-second (FPS) comparison in NewsFeedApp. Baseline: results with default flags; Size Optimized: results with `icf_safe` and `outline` on.

	Baseline (FPS)	Size Optimized (FPS)
Run 1	36.58	37.15
Run 2	36.00	35.41
Run 3	37.20	37.16
Average	36.59	36.57

4.3 Performance Impact

Since both `icf_safe` and `outline` passes introduce additional control flow instructions into the optimized code, it is important that they do not cause performance degradation. Here we compare several key performance metrics in mobile applications. The experiments are conducted on an iPhone SE2 with six CPU cores, 3 GB of RAM, and a quad-core GPU. We focus on NewsFeedApp since it contains a diverse set of use scenarios commonly found in mobile applications. The performance evaluation is conducted by appropriate profiling schemes provided by Apple's Xcode toolchain [38].

Application startup time. Application startup time is one of the key performance metrics for mobile applications. It measures the time between a user clicking on an application icon and the application finishing displaying its first frame after rendering. The startup delay directly translates into the wait time before a user can interact with the application. Reducing the startup delay can have a direct impact on user experience and significantly improve user engagement [39].

Table 3 presents startup delay impacts from our size optimizations on NewsFeedApp. The startup time is divided into the following four sequential phases: (i) library loading measures the time for an application to load the system's dynamic libraries; (ii) object loading measures the time for the application to load its initial objects; (iii) application launching includes logic, I/O, and memory access to set up various components and contents in an application, and (iv) initial frame rendering draws the first frame on the screen which also marks the completion of the startup process.

For both the baseline and optimized versions, we collect the average results after five identical runs. It is clear from the table that the size optimizations have a negligible impact on the startup time. This is mainly because the mobile applications we presented are mostly I/O bound, and the potential overhead caused by the new passes does not cause user-visible impact based on our profiling results.

Video playing performance. Video playing is another important use case in mobile applications. To measure the impact of our size optimizations on video playing performance, we collect the frames-per-second (FPS) metric using the video feed page in NewsFeedApp. For both the baseline and the optimized versions, we measure the average FPS over three one-minute long video play sessions using an automated script, where the script switches to the next available video every three seconds. The video feed algorithm randomly selects from a pool of available videos with similar characteristics. Table 4 shows the results of three runs and their average. We observe that both the baseline and our optimized versions have indistinguishable FPS numbers, indicating that the size optimization's impact on NewsFeedApp's video playing is negligible.

5 Related Work

Our work is closely related to work in compiler optimizations for code size reduction, link time and post-link-time optimizations, linker improvements, and optimizations for mobile applications.

Code size optimization. Rodrigo et al. [40, 41] explored various algorithms for finding similar functions and merging them to reduce application size. They use hash code to check function similarity which we also use to find identical functions. Lee et al. [30] and Chabbi et al. [29] perform code size optimizations for commercial iOS applications at the machine IR level during compilation, and they find that machine IR is a better target level than LLVM IR for their work. However, we perform optimizations on the machine instructions during linking. As a result, our technique reduces the build overhead.

More recently, machine learning-based approaches, such as MLGO [42] and CompilerGym [43] proposed to utilize machine learning models instead of heuristics. A combination of these approaches with our work might yield additional benefits in size reduction. Superpack [2] uses a custom compression algorithm to reduce Android bytecode [44] size. In contrast, this work focuses on native code using code transformations instead of compression. Moreover, this work complements several compiler optimizations [5–8, 11–16, 45] to further reduce the code size.

Link time and post link time optimization. Glek et al. [17] developed LTO for GCC [46] for performance and package size reduction. However, their approach is memory intensive and has scalability issues. Johnson et. al. [31] introduces ThinLTO, which is a lightweight LTO scheme that mostly runs in parallel and reduces both build time and memory usage. ThinLTO shows performance benefits similar to a full LTO approach, while its build time and memory consumption are smaller than that of the full LTO scheme. Several post-link-time optimizations [25, 26] leverage profile data for post-link optimizations for data center applications, while the focus of this work is mobile applications.

Linker improvement. Traditional linkers [47–49] focused on correctness, robustness, stability, and backward compatibility. Since link time is the dominating factor within modern rapid develop-build-debug cycles, newer linkers [49–51] instead focus on reducing the link time by utilizing parallel data structures and algorithms [51]. On the contrary, this work focuses on code size reduction through novel program analysis and optimizations conducted inside a linker.

Optimization for mobile applications. More generally, improvements for mobile applications cover language design and compiler optimizations. This includes size reduction [52, 53], stalled feature flags removal [54], improving Swift protocols [55], and so on. In addition, a large body of literature exists regarding performance optimizations for mobile applications, by improving responsiveness [56], memory management [57], state management [58], as well as startup time [39, 59]. Our work explores code size optimization which complements existing work.

6 Conclusion

In this work, we propose a novel framework for performing linker code size optimization for native mobile applications. It focuses on enhancing the linker with transformation passes that outline common code sequences and deduplicate identical functions. We reduce the binary size of three widely-used commercial iOS mobile applications by 18.4% on average, without any user noticeable performance degradations. Compared to existing LTO-based size optimizations, our work also significantly reduces build time overhead by confining the transformations within the linker, thus avoiding the need to piggyback on the compiler's optimization passes which tend to be prohibitively expensive for size optimization.

Future directions include improving the effectiveness of the outlining pass by optimizing for language-specific features, adding support for data section deduplication, using profiles to better guide transformation targets, and porting the optimizations to other linkers.

Acknowledgments

We would like to thank Yang Yu, Zhangjing Yuan, Kehong Huang and Chi Zhang for their help in testing and integrating this work in the applications. We thank Yuanshuo Zhu and Luchuan Guo for their support in this project. We appreciate Justin Wei and the anonymous reviewers' valuable comments and suggestions.

References

[1] Stephanie Chan. The iPhone's Top Apps Are Nearly 4x Larger Than Five Years Ago. https://sensortower.com/blog/ios-app-size-growth-2021, 2021. [Online; accessed 5-July-2022].

[2] Sapan Bhatia. Superpack: Pushing the Limits of Compression in Facebookś Mobile Apps. https://engineering.fb.com/2021/09/13/core-data/superpack/, 2022. [Online; accessed 5-July-2022].

[3] Apple App store. https://www.apple.com/app-store/, 2022. [Online; accessed 5-July-2022].

[4] Árpád Beszédes, Rudolf Ferenc, Tibor Gyimóthy, André Dolenc, and Konsta Karsisto. Survey of Code-Size Reduction Methods. *ACM Comput. Surv.*, 35(3):223–267, sep 2003. ISSN 0360-0300. doi:10.1145/937503.937504. URL https://doi.org/10.1145/937503.937504.

[5] Linda Torczon and Keith Cooper. *Engineering A Compiler*. Morgan Kaufmann Publishers Inc., 2007.

[6] John Cocke. Global Common Subexpression Elimination. *SIGPLAN Not.*, 5(7):20–24, jul 1970. ISSN 0362-1340. doi:10.1145/390013.808480. URL https://doi.org/10.1145/390013.808480.

[7] Robert Kennedy, Sun Chan, Shin-Ming Liu, Raymond Lo, Peng Tu, and Fred Chow. Partial Redundancy Elimination in SSA Form. 21(3): 627–676, may 1999. ISSN 0164-0925. doi:10.1145/319301.319348. URL https://doi.org/10.1145/319301.319348.

[8] Mark N. Wegman and F. Kenneth Zadeck. Constant Propagation with Conditional Branches. *ACM Trans. Program. Lang. Syst.*, 13(2):

181–210, apr 1991. ISSN 0164-0925. doi:10.1145/103135.103136. URL https://doi.org/10.1145/103135.103136.

[9] Alfred V Aho, Monica S Lam, Ravi Sethi, and Jeffrey D Ullman. *Compilers: Principles, Techniques, & Tools*. 2007.

[10] Marcelino Rodriguez-Cancio, Benoit Combemale, and Benoit Baudry. Automatic Microbenchmark Generation to Prevent Dead Code Elimination and Constant Folding. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 132–143, 2016.

[11] Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. Global Value Numbers and Redundant Computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 12–27, 1988.

[12] Jeremy Lau, Stefan Schoenmackers, Timothy Sherwood, and Brad Calder. Reducing Code Size with Echo Instructions. In *Proceedings of the 2003 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, CASES '03, page 84–94, New York, NY, USA, 2003. Association for Computing Machinery. ISBN 1581136765. doi:10.1145/951710.951724. URL https://doi.org/10.1145/951710.951724.

[13] Jens Ernst, William Evans, Christopher W. Fraser, Todd A. Proebsting, and Steven Lucco. Code Compression. In *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*, PLDI '97, page 358–365, New York, NY, USA, 1997. Association for Computing Machinery. ISBN 0897919076. doi:10.1145/258915.258947. URL https://doi.org/10.1145/258915.258947.

[14] Tobias J.K. Edler von Koch, Björn Franke, Pranav Bhandarkar, and Anshuman Dasgupta. Exploiting Function Similarity for Code Size Reduction. volume 49, page 85–94, New York, NY, USA, jun 2014. Association for Computing Machinery. doi:10.1145/2666357.2597811. URL https://doi.org/10.1145/2666357.2597811.

[15] Wen-Ke Chen, Bengu Li, and Rajiv Gupta. Code Compaction of Matching Single-entry Multiple-exit Regions. In *International Static Analysis Symposium*, pages 401–417. Springer, 2003.

[16] Henry Massalin. Superoptimizer: A Look at the Smallest Program. In *Proceedings of the Second International Conference on Architectual Support for Programming Languages and Operating Systems*, ASPLOS II, page 122–126, Washington, DC, USA, 1987. IEEE Computer Society Press. ISBN 0818608056. doi:10.1145/36206.36194. URL https://doi.org/10.1145/36206.36194.

[17] Taras Glek and Jan Hubicka. Optimizing Real World Applications with GCC Link Time Optimization. 2010.

[18] Joseph Caldwell and Shigeru Chiba. Reducing Calling Convention Overhead in Object-oriented Programming on Embedded ARM Thumb-2 Platforms. volume 52, pages 146–156. ACM New York, NY, USA, 2017.

[19] Saumya K Debray, William Evans, Robert Muth, and Bjorn De Sutter. Compiler Techniques for Code Compaction. volume 22, pages 378–415. ACM New York, NY, USA, 2000.

[20] Benjamin Schwarz, Saumya Debray, Gregory Andrews, and Matthew Legendre. PLTO: A Link-time Optimizer for the Intel IA-32 Architecture. In *Proc. 2001 Workshop on Binary Translation (WBT-2001)*, 2001.

[21] Bjorn De Sutter, Ludo Van Put, Dominique Chanet, Bruno De Bus, and Koen De Bosschere. Link-time Compaction and Optimization of ARM Executables. volume 6, pages 5–es. ACM New York, NY, USA, 2007.

[22] Dominique Chanet, Bjorn De Sutter, Bruno De Bus, Ludo Van Put, and Koen De Bosschere. System-wide Compaction and Specialization of the Linux Kernel. In *Proceedings of the 2005 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 95–104, 2005.

[23] Haifeng He, John Trimble, Somu Perianayagam, Saumya Debray, and Gregory Andrews. Code Compaction of an Operating System Kernel. In *International Symposium on Code Generation and Optimization (CGO'07)*, pages 283–298. IEEE, 2007.

[24] Nicolas Pitre. Shrinking the Kernel with Link-time Optimization. https://lwn.net/Articles/744507/, 2022. [Online; accessed 5-July-2022].

[25] Maksim Panchenko, Rafael Auler, Bill Nell, and Guilherme Ottoni. Bolt: A Practical Binary Optimizer for Data Centers and Beyond. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 2–14. IEEE, 2019.

[26] Google. PROPELLER: Profile Guided Optimizing Large Scale LLVM-based Relinker. https://github.com/google/llvm-propeller, 2019. [Online; accessed 8-August-2022].

[27] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.

[28] Jessica Paquette. Reducing Code Size Using Outlining. https://www.llvm.org/devmtg/2016-11/Slides/Paquette-Outliner.pdf, 2016. [Online; accessed 4-August-2022].

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

[30] Kyungwoo Lee, Ellis Hoag, and Nikolai Tillmann. Efficient Profile-Guided Size Optimization for Native Mobile Applications. CC 2022, page 243–253, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450391832. doi:10.1145/3497776.3517764. URL https://doi.org/10.1145/3497776.3517764.

[31] Teresa Johnson, Mehdi Amini, and Xinliang David Li. ThinLTO: Scalable and Incremental LTO. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 111–121, 2017. doi:10.1109/CGO.2017.7863733.

[32] LLVM Link Time Optimization: Design and Implementation. https://llvm.org/docs/LinkTimeOptimization.html. [Online; accessed 8-August-2022].

[33] Apple ld64 Linker. https://opensource.apple.com/source/ld64/, 2022. [Online; accessed 5-July-2022].

[34] Arm A64 Instruction Set Architecture. https://developer.arm.com/documentation/ddi0596/2021-09, 2021. [Online; accessed 4-August-2022].

[35] The DWARF Debugging Standard. https://dwarfstd.org/. [Online; accessed 8-August-2022].

[36] Writing ARM64 Code for Apple Platforms. https://developer.apple.com/documentation/xcode/writing-arm64-code-for-apple-platforms, 2022. [Online; accessed 10-November-2022].

[37] Sriraman Tallam, Cary Coutant, Ian Lance Taylor, Xinliang David Li, and Chris Demetriou. Safe ICF: Pointer Safe and Unwinding Aware Identical Code Folding in the Gold Linker. 2010.

[38] Apple Xcode. https://developer.apple.com/xcode/, 2022. [Online; accessed 16-August-2022].

[39] Tingxin Yan, David Chu, Deepak Ganesan, Aman Kansal, and Jie Liu. Fast App Launching for Mobile Devices Using Predictive User Context. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*, MobiSys '12, page 113–126, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450313018. doi:10.1145/2307636.2307648. URL https://doi.org/10.1145/2307636.2307648.

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

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

[42] Mircea Trofin, Yundi Qian, Eugene Brevdo, Zinan Lin, Krzysztof Choromanski, and David Li. MLGO: A Machine Learning Guided Compiler

Optimizations Framework. *arXiv preprint arXiv:2101.04808*, 2021.

[43] Chris Cummins, Bram Wasti, Jiadong Guo, Brandon Cui, Jason Ansel, Sahir Gomez, Somya Jain, Jia Liu, Olivier Teytaud, Benoit Steiner, Yuandong Tian, and Hugh Leather. CompilerGym: Robust, Performant Compiler Optimization Environments for AI Research. In *Proceedings of the 20th IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '22, page 92–105. IEEE Press, 2022. ISBN 9781665405843. doi:10.1109/CGO53902.2022.9741258. URL https://doi.org/10.1109/CGO53902.2022.9741258.

[44] Android. Dalvik Bytecode. https://source.android.com/devices/tech/dalvik/dalvik-bytecode, 2022. [Online; accessed 5-July-2022].

[45] Thomas J. Watson IBM Research Center. Research Division, FE Allen, and J Cocke. *A Catalogue of Optimizing Transformations*. 1971.

[46] GCC, the GNU Compiler Collection. https://gcc.gnu.org/, 2022. [Online; accessed 5-July-2022].

[47] Chamberlain Steve and Lance Taylor Ian. The GNU Linker. pages 1–154, 2022.

[48] Belousov Konstantin. BSD Linker. pages 1–20, 2011.

[49] Lance Taylor Ian. A New ELF Linker. In *2008 GCC Developers' Summit*, pages 30–36, 2008.

[50] Rui Ueyama. lld: A Fast, Simple and Portable Linker. In *LLVM Developer's Meeting*, 2017.

[51] Rui Ueyama. mold: A Faster Drop-in Replacement Linker for the Default GNU ld. 2022.

[52] ASM, Java Bytecode Manipulation and Analysis Framework. https://asm.ow2.io/, 2022. [Online; accessed 5-July-2022].

[53] Redex, A Bytecode Optimizer for Android Apps. https://github.com/facebook/redex/, 2022. [Online; accessed 5-July-2022].

[54] Murali Krishna Ramanathan, Lazaro Clapp, Rajkishore Barik, and Manu Sridharan. Piranha: Reducing Feature Flag Debt at Uber. In *2020 International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP 2020)*, 2020. doi:10.1145/3377813.3381350. URL files/ICSE20-SEIP-Piranha.pdf.

[55] Rajkishore Barik, Manu Sridharan, Murali Krishna Ramanathan, and Milind Chabbi. Optimization of Swift Protocols. *Proceedings of the ACM on Programming Languages (PACMPL), Volume 3, Issue OOPSLA*, 2019. doi:10.1145/3360590.

[56] Yu Lin, Cosmin Radoi, and Danny Dig. Retrofitting Concurrency for Android Applications through Refactoring. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, page 341–352, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450330565. doi:10.1145/2635868.2635903. URL https://doi.org/10.1145/2635868.2635903.

[57] Niel Lebeck, Arvind Krishnamurthy, Henry M Levy, and Irene Zhang. End the Senseless Killing: Improving Memory Management for Mobile Operating Systems. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 873–887, 2020.

[58] Umar Farooq, Zhijia Zhao, Manu Sridharan, and Iulian Neamtiu. LiveDroid: Identifying and Preserving Mobile App State in Volatile Runtime Environments. *Proceedings of the ACM on Programming Languages*, 4 (OOPSLA):1–30, 2020.

[59] Abhinav Parate, Matthias Böhmer, David Chu, Deepak Ganesan, and Benjamin M. Marlin. Practical Prediction and Prefetch for Faster Access to Applications on Mobile Phones. In *Proceedings of the 2013 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, UbiComp '13, page 275–284, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450317702. doi:10.1145/2493432.2493490. URL https://doi.org/10.1145/2493432.2493490.

Received 2022-11-10; accepted 2022-12-19