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 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\(^1\), each with hundreds of millions of daily active users. Our technique on average achieves 18.4% binary size reduction across the three commercial applications without any user-perceivable performance degradations.

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 [6]. In addition to performance benefits, many popular compiler optimizations may also reduce code size, such as dead and unreachable code elimination [7], common sub-expression elimination [8], partial redundancy elimination [9], constant and copy propagation [10, 11], constant folding [12], value numbering [13], register allocation and instruction scheduling [14], code compression [15–17], and peephole optimizations [18]. Other than compiler optimizations, link-time optimizations (LTO) and post-link-time optimizations [19–28] have also shown success in reducing code size.

State of the art. iOS applications are commonly compiled using LLVM [29], 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 [30]. 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

\(^1\)We do not use the actual names of commercial applications for anonymity.

\(^2\)This technique triples the build time according to [4].
To address these challenges, we propose a novel technique 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.

\(^3\)Embedding LLVM IR in the object files requires explicitly specifying -fembed-bitcode option, which is off by default in Clang.

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, and finally we conclude this work in Section 6.

## 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) link time 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 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. Byte-wise 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 pivot instructions to be the branch targets of control flow instructions and the start/end/jump targets in an exception handling table. Removing such pivot 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 pivot instructions is preserved upon discovery and reused by various optimization passes during linking. Figure 2 shows an example of pivot instructions. The two highlighted ld64 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.

```
104a5f0fc: mov x20, x0
104a5f00: 1dr x0, [x19, #72]
104a5f04: cbzx0, 0x104a5f18
104a5f08: 1dr x22, [x0]  
```

Figure 2. An example of instruction visibility where the two highlighted ld64 instructions are the branch targets of the corresponding cbz/cbnz instructions. They are marked as visible.

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 [4]. We instead propose and implement outlining inside a linker for...
mov x21, x0
mov x0, x19
mov x1, x20

(a) b <objc_msgSend>
(b) b <-isPlatformVersionAtLeast>
mov w3, #0
mov x1, x20
mov x0, x19
mov x21, x0

objc_msgSend
d2, d3, [x8, #16]

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.

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 or even hours [4]. 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 is capable of analyzing the whole program. For example, our linker outliner can optimize third-party libraries available only in binary format, which is beyond the capability of an LLVM IR-level 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 uses of 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 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.

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
            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)

        // 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
        foreach callSites ∈ collectCallSites(hashToCallSites) do
            foreach outlineSequence ∈ callSites do
                // replace to-be-outlined sequences with branching logic
                createBranchingLogic(outlineSequence)
                // update branch target indices for control flow instructions
                updateBranchTarget(outlineSequence)

                // update metadata (i.e., exception handling table) of the function
                updateMetadata(outlineSequence)

                updateMetadata(outlineSequence)
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 original binary.

**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 are able to 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 1d64 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 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 correct behavior when an exception happens and the corresponding stack unwinds. 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 the 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 1d64 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 where the modifications are applicable. 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.\(^5\)

\(^5\)We note that a prior work [5] proposed a frame code optimization technique during LLVM machine IR optimization, while we conduct frame code outlining during linking.
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
sub sp, sp, #320

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.

Figure 5(a) 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 and 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 5(a), 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 registers 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 5(b) 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, the normalized frame code sequences from different functions can share the same outlined frame-code function as long as 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 unique copy. 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.

948965a4 <-{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 <-{AudioPlayerDatasetIsCommunity:}:
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 exact 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 commonly used arithmetic operations is equality comparison. It checks whether two function pointers point to the same address (i.e., the
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.
(b) Illustration of assembly code generated by icf_safe.

Figure 7. Handling function pointer comparisons in icf_safe.

function’s implementation). Figure 7(a) 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=a11 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. The icf_safe option adds redirection logic to preserve correctness under function pointer equality comparison. Figure 7(b) illustrates 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=a11 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.

3 Implementation

We implement previously described ICF and outlining optimizations as additional passes in the Apple ld64 linker (version 609) [33]. Our implementation logic is 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. The ld64 linker schedules these new optimization passes 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 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 $1 \times 10^6$ to $2 \times 10^6$ 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 for the applications, build time comparisons, and comparison between our technique and full LTO based approach. Furthermore, we compare performance impacts with and without our optimizations.

4.1 Code Size Reduction

Figure 8 shows the size reduction for each application with a baseline design pointing to its respective production build configuration. 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 -O2 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 [4, 5]. In addition, our results agree with [5] 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 difference 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% 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.

4.3 Comparison with a Full LTO Based Approach

As discussed in Section 1, full LTO based machine outlining can also achieve significant size savings at the cost of increased build time. Here we compare our approach with the full LTO based approach over size savings and build time increase, using the ShortVideoApp as a study vehicle. Note that in the full LTO based approach, we can vary the number of files participating in LTO by selectively enabling/disabling -flto, an LLVM compiler flag, over individual files during compilation.

Figure 9 shows the details of this comparison. The baseline approach uses the same compilation options as the baseline approach in Section 4.1. Our approach enables the two size optimization passes on top of the baseline. Collectively our passes help achieve a 26 MB binary size reduction with a build time increase of 144 seconds. The three full LTO based runs enable full LTO on the five largest modules out of roughly the 300 modules in the application, with increasing size saving options. These additional size saving options include the --enable-machine-outliner-always and --machine-outliner-reruns=1 flags. These are LLVM compiler flags that can alter the default behaviors of its machine outlined. We did not report results with --machine-outliner-reruns larger than 1 because it leads to diminishing return in size (<0.5%) and further increases build time. We note that some modules are compiled with -Oz flag, which automatically enables the compiler’s machine outlining pass.
Table 2. Build time profile on NewsFeedApp, ShortVideoApp and CollaborationSuiteApp and the percentage overhead of our passes. m: minutes, s: seconds.

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

Figure 9. Comparisons of build time overhead vs. binary size reduction between our approach and full LTO based machine outlining approach for ShortVideoApp. Baseline: Default approach without full LTO/linker size optimization. Ours: Our approach with icf_safe and outline passes. FullLTO-1: Turn on full LTO on five largest components of the application. FullLTO-2: Adding --enable-machine-outliner=always on top of FullLTO-1. FullLTO-3: Adding --machine-outliner-reruns=1 on top of FullLTO-2.

While enabling full LTO on more modules would lead to increased size savings, we were unable to enable full LTO over all modules due to frequent out-of-memory errors and build time surpassing 10 hours.

We observe that the full LTO based approach can indeed effectively reduce the binary size. However, the overall gain is significantly lower. Moreover, the build time increase for the full LTO based approach is significantly higher at 1.5 to 2.5 hours compared to less than 10 minutes in our approach. This is mainly because the full LTO based approach needs to execute a significant portion of the IR optimizations and all backend passes on the merged module using a single thread. This can potentially cause major release build turnover issues and put both programmer productivity and development efficiency at risk. Figure 9 demonstrates that our approach offers a better proto-optimal tradeoff between size reduction and build time overhead than the full LTO-based approach.

4.4 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 actually 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 due to the fact that 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 having 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.

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

<table>
<thead>
<tr>
<th>Run</th>
<th>Baseline (FPS)</th>
<th>Size Optimized (FPS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Run 1</td>
<td>36.58</td>
<td>37.15</td>
</tr>
<tr>
<td>Run 2</td>
<td>36.00</td>
<td>35.41</td>
</tr>
<tr>
<td>Run 3</td>
<td>37.20</td>
<td>37.16</td>
</tr>
<tr>
<td>Average</td>
<td>36.59</td>
<td>36.57</td>
</tr>
</tbody>
</table>

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 [7–10, 13–18, 45] to further reduce the code size.

**Link time and post link time optimization.** Glek et al. [19] developed link time optimizations (LTO) for GCC [46] for performance and package size reduction. However, their approach remains resource 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 runtime and resource overhead. ThinLTO shows performance benefits similar to a full LTO approach, while its build time and resource consumptions are smaller than that of the full LTO scheme. Several post-link-time optimizations [27, 28] 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 aggressively utilizing parallel data structures and algorithms, including even a complete redesign that heavily favors link speed [51]. On the contrary, this work focuses on code size reduction through novel program analysis and optimizations conducted entirely inside a linker.

**Optimization for mobile applications.** More generally, improvements for mobile applications cover language design and intermediate code 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 could complement existing work when used in combination.

### 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 new transformation passes that outline common code sequences and deduplicate
identical functions. We reduce the binary size of three widely used large-scale 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 in 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 popular linkers and widely used mobile applications.

References


