Scalable Size Inliner for Mobile Applications (WIP)

Kyungwoo Lee
Meta
Menlo Park, CA, USA
kyulee@fb.com

Manman Ren
Meta
Menlo Park, CA, USA
mren@fb.com

Shane Nay
Meta
Menlo Park, CA, USA
snay@fb.com

Abstract

Inlining is critical for both performance and size in mobile apps. When building large mobile apps, ThinLTO, a scalable link-time optimization is imperative in order to achieve both optimal size and build scalability. However, inlining with ThinLTO is not tuned to reduce the code size because each module inliner works independently without modeling the size cost across modules, and functions are often not eligible to import due to private references, appearing in Objective-C or Swift for iOS. This paper extends the bitcode summary to perform a global inlining analysis to find inline candidates for saving the code size. Using this summary information, a pre-inliner eagerly inlines the candidates that are proven to shrink the size. When the inline candidates are not eligible to import, a pre-merger combines their bitcode modules to remove inline restrictions. Our work improves the size of real-world mobile apps when compared to the *MinSize* (*-Oz*) optimization level. We reduced the code size by 2.8% for `SocialApp` and 4.0% for `ChatApp`.

CCS Concepts: • **Software and its engineering** → **Compilers**; • **Computer systems organization** → *Embedded systems.*

Keywords: inlining, ThinLTO, size optimization, mobile applications, iOS

ACM Reference Format:
Kyungwoo Lee, Manman Ren, and Shane Nay. 2022. Scalable Size Inliner for Mobile Applications (WIP). In *Proceedings of the 23rd ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '22), June 14, 2022, San Diego, CA, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3519941.3535074

1 Introduction

Mobile app size continues to grow as new features are constantly added to meet users' needs [5, 12]. They are mostly optimized for size while keeping the performance of start-up and key scenarios [11]. Outlining [4, 8, 11, 15–17] improves the code size by factoring out the common code into functions. Conversely, inlining [3, 14] has been primarily considered a speed optimization which removes call overhead at the cost of cloning the callee, and provides a bigger scope for other optimizations to be effective. Inlining is also important for size, even at *MinSize* (*-Oz*) optimization level [6, 22].

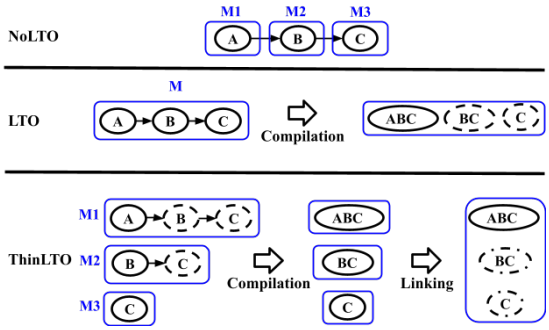

Figure 1. Inlining for size in LTO and ThinLTO. Functions *A*, *B*, and *C* are defined in modules *M*1, *M*2, and *M*3, respectively. LTO merges all modules to inline *B* and *C* into *A* and deletes their bodies at compile time. ThinLTO imports the inline candidates for each module that runs independently. The linker can dead-strip unreferenced functions, *BC* or *C*.

Commercial mobile apps [4, 11] are globally optimized at link time using either the regular *full* link-time optimization (LTO) or ThinLTO [10]. When building large apps, we use ThinLTO [10] because using LTO is impractical, taking a long time to build. Figure 1 shows how LTO or ThinLTO can save the code size by inlining functions. Three functions *A*, *B*, and *C* are defined in modules *M*1, *M*2, and *M*3, respectively. The call edges are represented as arrows between functions. Functions *B* and *C* cannot be inlined across modules with NoLTO. LTO merges the whole modules to inline *B* and *C* while deleting them at compile time. Unlike LTO that can analyze the entire intermediate representation (IR), ThinLTO uses the combined bitcode summary to import the inline candidates for each module compilation. The final size can be reduced at link time by dead-stripping unreferenced functions.

This inline framework with ThinLTO has the following limitations toward size optimization: (i) The bitcode summary keeps one call edge for each callee, even if there are multiple call sites to the callee. The function import and inlining decisions are largely tuned for speed, but not for

LCTES '22, June 14, 2022, San Diego, CA, USA

ACM ISBN 978-1-4503-9266-2/22/06.
https://doi.org/10.1145/3519941.3535074

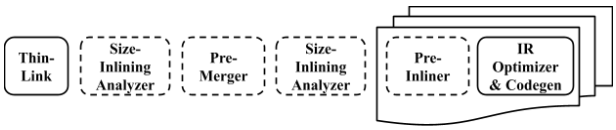

Figure 2. Overview of size inliner passes (in the dashed boxes) with ThinLTO. *Size inlining analyzer* determines inline candidates at the summary level. *Pre-merger* groups bitcode modules and merges each group to remove inline restrictions. Again, *size inlining analyzer* with a new set of modules guides *pre-inliner* to inline the candidates ahead.

size. In addition, these decisions are made independently per module. (ii) An inline candidate may not be imported inherently, and thus cannot be inlined across modules. For instance, if a function has references privately bound within a module, it is not eligible to import.

This paper addresses these shortcomings by modeling the size cost across modules followed by *pre-inliner* and *pre-merger*. Using these techniques, we improved the code size (i.e., text section size) by 2.8% for `SocialApp`, and 4.0% for `ChatApp` compared to *-Oz* optimization with ThinLTO. In particular, this paper makes the following contributions:

1. We propose *pre-inliner* which secures inlining for size using an inter-module *size inlining analyzer* at the summary level.
2. We describe *pre-merger* which regroups bitcode modules to lift inline barriers across modules.
3. We evaluate the aforementioned techniques with two real-world mobile apps and an open-source compiler, `Clang`.

The rest of this paper is organized as follows. Section 2 proposes our size inliner framework with ThinLTO. Section 3 presents our evaluation. Section 4 discusses related and future work, and Section 5 concludes the paper.

2 Size Inliner

LTO inliner makes inlining decisions with the entire IR, which can be optimal. ThinLTO inliner imports cross-module inline candidates, and makes inlining decisions independently for each module. Figure 3 shows how the ThinLTO inliner may decide to inline function D into function B in $M2$ as well as function C in $M3$. If the size for D is large, inlining D into two call sites may increase the overall size.

Figure 2 shows an overview of our size inliner passes with ThinLTO. After *ThinLink*, which combines the bitcode summary, *size inlining analyzer*, described in Section 2.1, globally determines inline candidates at the summary level. We will describe how to implement *pre-inliner* for each module in Section 2.2. For those inline candidates that are not eligible to import, *pre-merger* efficiently links their modules to expand inline scope, as discussed in Section 2.3.

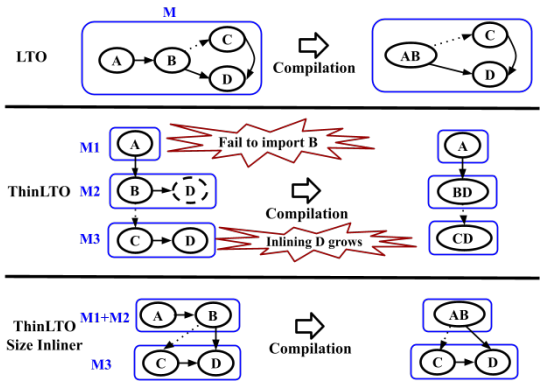

Figure 3. Inlining for size in LTO, ThinLTO, and our size inliner. Function B dynamically calls function C via *objc_msgSend* [1], shown as a dotted arrow. Using the entire IR, LTO optimally inlines a single callee B into A but does not inline D. ThinLTO fails to import B into $M1$ since B has private references within $M2$. ThinLTO may inline both Ds in $M2$ and $M3$, increasing the overall size. Our size inliner merges $M1$ and $M2$ to remove inline restrictions on B. D is not inlined using a global inlining analysis.

2.1 Size Inlining Analyzer

We now consider a simplified size cost model for inlining that has the total number of call sites for a given callee, using the global call graph at the summary level. We ignore the additional size benefit from downstream optimizations on the inlined code. We also assume the callee will be completely inlined into all call sites and thus removed.

$$\begin{aligned} C_{before} &= C_{callee} + N \cdot C_{call} \\ C_{after} &= C_{callee} \cdot N \\ &\text{where } N \text{ is the number of call sites,} \\ &\quad C_{callee} \text{ is the callee size, and} \\ &\quad C_{call} \text{ is the call overhead.} \end{aligned} \tag{1}$$

When optimizing for size, we should inline as long as C_{after} is smaller than C_{before}. When $N = 1$, inlining always decreases the size. For the case of $N \geq 2$, we iteratively make inlining decisions by updating the call graph at the summary level. Similar to Figure 3, if function D is completely inlined to both function B and C, then D will be removed from the call graph while B will become BD and C will become CD. Note all the call sites of D will appear in BD and CD. We can find all the inlining candidates using a greedy algorithm by handling the lowest N first, until inlining increases the size and $C_{after} > C_{before}$.

The current bitcode summary with ThinLTO conservatively represents a single call edge for a pair of caller and callee by aggregating multiple call sites. This, in turn, prevented us from computing the size cost precisely in Equation 1. We extended *CalleeInfo*, attached to each call edge,

to include the number of call sites to a callee. N, the total number of call sites to the callee, can be computed from the call graph.

Our initial implementation only handles the case of $N = 1$, which is absolutely beneficial for size, which is assumed for the rest of paper. However, we handled several practical issues to make inlining effective. First, some functions are dynamically exported or accessed by the regular objects outside the scope of bitcode modules. These call edges cannot be modeled in this analysis, but the linker conservatively preserves those functions, therefore inlining them increases the size. We conveyed such symbol resolution information from the linker to remove them from the inline candidates. We also bailed out functions with user inline attributes like *__attribute__((noinline))* or *__attribute__((always_inline))* to avoid unnecessary interactions. Finally, when functions are large, the size reduction from inlining might be smaller than extra spill code from high register pressure. Empirically, we excluded functions whose IR size was greater than 500.

2.2 Pre-Inliner

An inliner typically prioritizes the small inline candidates that are frequently called. The existing inliner's heuristics with ThinLTO are hard to tune when optimizing for size. Therefore, we propose adding *pre-inliner* that imports and inlines candidates for each module compilation, shown in Figure 2. *Pre-inliner* extends the existing *always-inliner* [19] and performs the published decisions from *size inlining analyzer* described in Section 2.1.

In order to preclude adversary interactions, we must execute out the published inlining decisions prior to the existing inliner. For instance, function B is to be inlined into A because of a single call edge in Figure 3. Although the callers of A do not appear in this figure, if A were inlined into many call sites from other heuristics before B being inlined into A, the number of call sites to B would become larger. Forcing inlining B to the already inlined instances of A would increase the size. As expected, our prototype handled pre-inlining candidates as if they were with *__attribute__((always_inline))* when compiling `Clang`, and the code size was increased by 1.7%. We will present the details on the size saving with *pre-inliner* in Section 3.2.

2.3 Pre-Merger

When a cross-module inline candidate has references privately bound in a module, it is not eligible to import, and thus we cannot inline it with ThinLTO. In Figure 3, function B in module $M2$ is uniquely called from function A in module $M1$. However, function B in module $M2$ may access private Objective-C metadata to dynamically dispatch a call to C in $M3$. In this case, we pre-merge $M1$ and $M2$ to produce $M1 + M2$ to allow inlining B into A within the same module.

For those inline candidates computed in the *size inlining analyzer* described in Section 2.1, if they are not eligible to import, *pre-merger* uses a union-find data structure [18] to recursively link their corresponding bitcode modules. The bitcode summary is recomputed for each merged module, and the new set of modules are published. Then the remaining ThinLTO passes follow.

Pre-merging many modules may unbalance the size of ThinLTO modules, increasing the overall build time. To limit the build time increase, we used a tunable parameter (*merge threshold*) that controls the maximum number of merged modules. We chose the default value to be 500. To realize more size wins with fewer merges, we prioritized merging of two modules with the decreasing order of *inline affinity*. We define *inline affinity* for two modules as the number of cross-module inline candidates in between them. Modules with higher *inline affinity* get merged first. We will present the details on the trade-off of code size and build time in Section 3.3.

Direct annotations for methods [20] in Objective-C tell the compiler to use static dispatch instead of dynamic dispatch. These direct methods often have private references to Objective-C metadata, thus becoming ineligible to be imported across modules. *Pre-merger* removes such inline restrictions on the direct methods. At our company, we regularly perform whole-app analysis offline to find direct method candidates. Codemod service [7] automatically generates source patches with annotations.

3 Evaluation

3.1 Benchmark

Table 1 summarizes our benchmark. `SocialApp` is one of the largest non-gaming mobile apps with dozens of dynamically loaded libraries (dylibs). Direct annotations for methods are enabled in `SocialApp`, as described in Section 2.3. This app is written using a mix of Objective-C and Swift. `ChatApp` is a medium sized mobile app with a mix of Objective-C and C++. `Clang` is an open-source compiler, using the 12.0 release. Unlike the other two mobile apps, `Clang` is computationally intensive, so we evaluated size as well as performance. In particular, we measured the runtime performance of libLTO [9], called from the linker (LD), when natively building another libLTO. All benchmarks were compiled with ThinLTO and the baseline was built with *-Oz*, unless otherwise specified.

Table 1. Statistics of Applications used for Evaluation.

App	Code Size	Direct Annot.	Language	OS
`SocialApp`	106 MB	Yes	Obj-C/Swift	iOS
`ChatApp`	33 MB	No	Obj-C/C++	iOS
`Clang`	12 MB	No	C/C++	MacOS

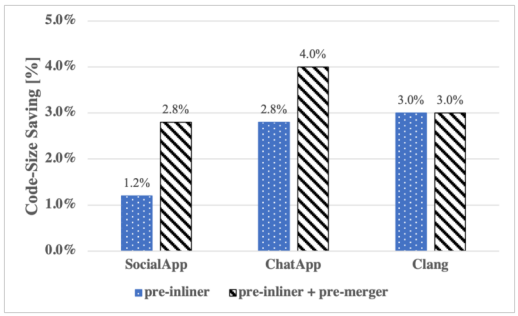

Figure 4. Code-size saving with *pre-inliner* and *pre-inliner+pre-merger*. The code-size saving is calculated by 1 minus the ratio of each case over the baseline.

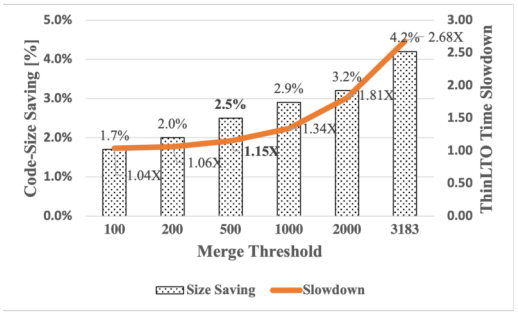

Figure 5. Code-size saving and ThinLTO build time with varying *merge threshold* for the largest binary of `SocialApp`.

3.2 Code Size Impact

Figure 4 compares the code-size improvement for `SocialApp`, `ChatApp`, and `Clang`. *Pre-inliner* improved the code size by 1.2% for `SocialApp`. Adding *pre-merger* improved the code size further by 2.8% because there are many direct methods in this app, as described in Section 2.3, that were not eligible to import before. Similarly, the code size was improved for `ChatApp` by 2.8% with *pre-inliner* and 4.0% with *pre-inliner* combined with *pre-merger*. `ChatApp` already has a large portion of C/C++ functions with private references to Objective-C metadata, similar to direct methods. `Clang` is a C/C++ program that is almost always eligible to import. Therefore, *pre-merger* is ineffective, but *pre-inliner* itself already reduced the code size by 3% for `Clang`.

3.3 Trade-Off

Figure 5 shows the code-size saving and the ThinLTO build time increase as *merge threshold* increases for the largest binary of `SocialApp`. We set the default *merge threshold* to 500, which increased the ThinLTO build time by 1.15*X* while saving the code size by 2.5%. Most binaries are much smaller than this binary, and they rarely reach this threshold.

We compared the code-size saving and speed-up with or without the machine outliner [13]. Note *-Oz* enables the machine outliner by default for AArch64 [2] we target. As shown in Table 2, without the outliner, the size and performance impacts from inlining were significant: 41.9% and 2.95*X*, for each. Expectedly, the outliner reduced the code size further at the cost of performance due to additional call overhead. Importantly, our size inliner improved both size and performance regardless of the outliner's presence. In short, when compared to *-Oz* with the default inliner and outliner, our size inliner produced 3% smaller and 6.1% faster `Clang`.

Table 2. Code-size saving and speed-up with or without the machine outliner [13] for Clang. The baseline disables the inliner (*-fno-inline-functions* [21]).

	Code-Size Saving	Speed-up
NoOutliner (**Default Inliner**)	41.9%	2.95X
NoOutliner (**Size Inliner**)	44.0%	3.11X
Outliner (**Default Inliner**)	46.9%	2.15X
Outliner (**Size Inliner**)	48.5%	2.28X

4 Discussion

Trofin et al. [22] proposed a machine learning guided inliner for size. Their numbers were collected with NoLTO, and they did not address ThinLTO-specific inlining issues. Our approach focuses on ThinLTO inlining by improving inlining scope and making inlining decisions using a global call graph.

Damásio et al. [6] targeted inlining for code-size reduction, similar to the method we used in this study. They simplified the inlinee's IR ahead to precisely model the size cost. We performed inlining decisions at the summary level. We also dealt with the inline candidates that were ineligible with ThinLTO.

We will extend our size inliner with $N \geq 2$, as described in Section 2.1. Instead of only targeting size, we will improve the heuristics in the inlining analyzer to find inline candidates for speed with size constraints. We can define *inline budget* as the percentage of size increase on top of the initial computational cost, C_{before}, and use the *inline budget* to adjust the trade-off of the speed and size. Once we have exploited all the inline candidates for size, we can continue finding candidates using profile or other performance-centric priority, until the *inline budget* is exhausted.

5 Conclusion

We presented our size inliner framework that globally models the size cost by extending the bitcode summary. We proposed *pre-inliner* to secure the size win, and *pre-merger* to overcome inline restrictions with ThinLTO. Our work improved the code size, 2.8% for `SocialApp` and 4.0% for `ChatApp`. When compared to the state-of-the-art, *-Oz* optimization with ThinLTO, `Clang` became 3% smaller and 6.1% faster.

References

[1] Apple. 2021. *Document for objc_msgSend.* https://developer.apple.com/documentation/objectivec/1456712-objc_msgsend

[2] Arm. 2021. Arm A64 Instruction Set Architecture. https://developer.arm.com/documentation/ddi0596/2021-09

[3] J. Michael Ashley. 1997. The Effectiveness of Flow Analysis for Inlining. *SIGPLAN Not.* 32, 8 (aug 1997), 99–111. https://doi.org/10.1145/258949.258959

[4] Milind Chabbi, Jin Lin, and Raj Barik. 2021. An Experience with Code-Size Optimization for Production IOS Mobile Applications. In *Proceedings of the 2021 IEEE/ACM International Symposium on Code Generation and Optimization* (Virtual Event, Republic of Korea) *(CGO '21)*. Association for Computing Machinery, New York, 363–366. https://doi.org/10.1109/CGO51591.2021.9370306

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

[6] Thaís Damásio, Vinícius Pacheco, Fabrício Goes, Fernando Pereira, and Rodrigo Rocha. 2021. *Inlining for Code Size Reduction.* Association for Computing Machinery, New York, 17–24. https://doi.org/10.1145/3475061.3475081

[7] Facebook. 2015. *Codmod: A tool/library to assist you with large-scale codebase refactors.* https://github.com/facebookarchive/codemod

[8] LLVM Compiler Infrastructure. 2021. MergeFunctions pass, how it works. https://llvm.org/docs/MergeFunctions.html

[9] LLVM Compiler Infrastructure. 2022. LLVM Link Time Optimization: Design and Implementation. https://llvm.org/docs/LinkTimeOptimization.html#liblto

[10] Teresa Johnson, Mehdi Amini, and Xinliang David Li. 2017. ThinLTO: Scalable and Incremental LTO. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization* (Austin) *(CGO '17)*. IEEE Press, 111–121.

[11] Kyungwoo Lee, Ellis Hoag, and Nikolai Tillman. 2022. Efficient Profile-Guided Size Optimization for Native Mobile Applications. In *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction (CC '22)*. https://doi.org/10.1145/3497776.3517764

[12] Meta. 2021. *Superpack: Pushing the limits of compression in Facebook's mobile apps.* https://engineering.fb.com/2021/09/13/core-data/superpack/

[13] Jessica Paquette. 2016. Reducing Code Size Using Outling. https://llvm.org/devmtg/2016-11/Slides/Paquette-Outliner.pdf

[14] Aleksandar Prokopec, Gilles Duboscq, David Leopoldseder, and Thomas Würthinger. 2019. An Optimization-Driven Incremental Inline Substitution Algorithm for Just-in-Time Compilers. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization* (Washington, DC, USA) *(CGO 2019)*. IEEE Press, 164–179.

[15] River Riddle. 2018. IR Outliner Pass. https://reviews.llvm.org/D53942

[16] Rodrigo C. O. Rocha, Pavlos Petoumenos, Zheng Wang, Murray Cole, Kim Hazelwood, and Hugh Leather. 2021. HyFM: Function Merging for Free. In *Proceedings of the 22nd ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems* (Virtual, Canada) *(LCTES 2021)*. Association for Computing Machinery, New York, 110–121. https://doi.org/10.1145/3461648.3463852

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

[18] Robert Endre Tarjan. 1975. Efficiency of a Good But Not Linear Set Union Algorithm. *J. ACM* 22, 2 (apr 1975), 215–225. https://doi.org/10.1145/321879.321884

[19] The Clang Team. 2022. Attributes in Clang: always_inline. https://clang.llvm.org/docs/AttributeReference.html#always-inline-force-inline

[20] The Clang Team. 2022. Attributes in Clang: objc_direct. https://clang.llvm.org/docs/AttributeReference.html#objc-direct

[21] The Clang Team. 2022. Clang command line argument reference. https://clang.llvm.org/docs/ClangCommandLineReference.html#compilation-flags

[22] Mircea Trofin, Yundi Qian, Eugene Brevdo, Zinan Lin, Krzysztof Choromanski, and David Li. 2021. MLGO: a Machine Learning Guided Compiler Optimizations Framework. *CoRR* abs/2101.04808 (2021). arXiv:2101.04808 https://arxiv.org/abs/2101.04808