Efficient Profile-Guided Size Optimization for Native Mobile Applications

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

Ellis Hoag
Meta
Menlo Park, CA, USA
ellishoag@fb.com

Nikolai Tillmann
Meta
Menlo Park, CA, USA
nikolait@fb.com

Abstract

Positive user experience of mobile apps demands they not only launch fast and run fluidly, but are also small in order to reduce network bandwidth from regular updates. Conventional optimizations often trade off size regressions for performance wins, making them impractical in the mobile space. Indeed, *profile-guided optimization* (PGO) is successful in server workloads, but is not effective at reducing size and page faults for mobile apps. Also, profiles must be collected from instrumenting builds that are up to 2*X* larger, so they cannot run normally on real mobile devices.

In this paper, we first introduce *Machine IR Profile* (MIP), a lightweight instrumentation that runs at the machine IR level. Unlike the existing LLVM IR instrumentation counterpart, MIP withholds static metadata from the instrumenting binaries leading to a 2/3 reduction in size overhead. In addition, MIP collects profile data that is more relevant to optimizations in the mobile space. Then we propose three improvements to the LLVM machine outliner: (i) the *global outliner* overcomes the local scope of the machine outliner when using ThinLTO, (ii) the *frame outliner* effectively outlines irregular prologues and epilogues, and (iii) the *custom outliner* outlines frequent patterns occurring in Objective-C and Swift. Lastly, we present our PGO that orders hot start-up functions to minimize page faults, and controls the size optimization level (*-Os* vs *-Oz*) for functions based on their estimated execution time driven from MIP. We also order cold functions based on similarity to minimize the compressed app size.

Our work improves both the size and performance of real-world mobile apps when compared to the *MinSize* (*-Oz*) optimization level: (i) in `SocialApp`, we reduced the compressed app size by 5.2%, the uncompressed app size by 9.6% and the page faults by 20.6%, and (ii) in `ChatApp`, we reduced them by 2.4%, 4.6% and 36.4%, respectively.

CC '22, April 02–03, 2022, Seoul, South Korea

ACM ISBN 978-1-4503-9183-2/22/04.
https://doi.org/10.1145/3497776.3517764

CCS Concepts: • **Software and its engineering** → **Compilers**; *Runtime environments*; • **Computer systems organization** → Embedded software; • **General and reference** → Performance.

Keywords: profile-guided optimizations, size optimizations, machine outlining, mobile applications, iOS

ACM Reference Format:
Kyungwoo Lee, Ellis Hoag, and Nikolai Tillmann. 2022. Efficient Profile-Guided Size Optimization for Native Mobile Applications. In *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction (CC '22), April 02–03, 2022, Seoul, South Korea.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3497776.3517764

1 Introduction

Mobile app size grows rapidly as new features are constantly added to meet user's needs. This growth conflicts with the need to minimize the network bandwidth from regular app updates, the size limitations imposed by app stores, and the limited available disk space on user's devices. The initial app startup time and the smoothness of interactions impacts user experience and directly correlates to user retention and thus affects business revenue. Mobile apps are mostly IO-bound because of interactions with various peripherals during the lifetime of the app, so there is less need to optimize for performance and they are generally optimized for size [4, 14]. Nonetheless, the performance of start-up and a few key scenarios are critical for mobile apps' success. For instance, `ChatApp` must launch chat threads quickly and minimize the latency of sending and receiving messages in real-time.

Profile-guided optimization (PGO) is beneficial for long running applications like server workloads for speed optimization by improving CPU and cache performance using function inlining, loop optimization, and block ordering. Applying traditional PGO in the same way poses several challenges to both instrumentation and optimization in the mobile space.

Instrumentation typically injects probes for control-flow edges and indirect call-targets and also injects associated metadata such as function names and probe layouts. The total instrumenting binary size is up to 2*X* the original size, which is too big and slow to distribute and run on a large number of representative end-user mobile devices. Sample-based PGO [5, 23] using hardware sample counters provided by modern

hardware can eliminate the need for instrumentation, but mobile platforms may not allow access to such hardware samples in user processes for security reasons. It is also impractical to store large volumes of sample data in the limited backing store on a mobile device or to send it over the network because of concerns about bandwidth and battery power.

The machine code characteristics and optimization objectives for mobile apps–and for iOS in particular–are drastically different from traditional server workloads. Table 1 presents the number of blocks per function in each percentile, and shows that functions in mobile apps have few blocks. For example, in `SocialApp` (which will be evaluated in Section 5), 75% of functions have only 3 blocks. Function inlining usually increases function basic block count, but in the mobile space inlining is constrained because (i) mobile apps are optimized for size and (ii) calls are mostly dynamically dispatched in Objective-C. Furthermore, Objective-C and Swift use many helper calls to support automatic reference counting (ARC) [12, 33]. From our measurements, 25% of the instructions in our mobile apps are call instructions (`bl` in the AArch64 architecture [1]), compared to only 4.5% in a speed optimized (*-O3*) `Clang` binary.

To the best of our knowledge, this is the first paper that efficiently applies a whole PGO pipeline to real-world native mobile apps and achieves better size and performance characteristics than *-Oz* optimization. We improved the compressed app size by 5.2%, the uncompressed size by 9.6% and the start-up performance by 20.6% for `SocialApp`, and improved them by 2.4%, 4.6% and 36.4%, respectively, for `ChatApp`. In particular, this paper makes the following contributions:

1. We propose a new lightweight instrumentation (MIP) [10] that is critical for mobile apps.
2. We describe three outliners on top of the existing LLVM machine outliner [24]: the global outliner [19], the frame outliner [18], and the custom outliner.
3. We describe how we apply PGO profiles for mobile apps:
   - We order functions to improve performance and compressed binary size.
   - We assign the best optimization level, e.g., *OptSize* (*-Os*), to each function individually.
4. In addition to evaluating the above techniques with two real-world mobile apps, `SocialApp` and `ChatApp`, we extensively compare performance, size, build-time, and compression rate with an open-source compiler, `Clang`.

The rest of this paper is organized as follows. Section 2 proposes MIP instrumentation. Section 3 describes our aggressive outlining techniques. Section 4 describes our optimization strategies with regard to hot or cold function ordering, and profile-driven size optimization settings. Section 5 presents our evaluation. Section 6 discusses related work and Section 7 concludes the paper.

Table 1. Block count per function in percentile.

App	p50	p75	p95	p99
`SocialApp`	1	3	11	29
`ChatApp`	3	6	24	70
`Clang` (*-Oz*)	3	6	23	92
`Clang` (*-O3*)	12	30	136	600

2 Instrumentation

In this section we introduce MIP [10], a novel lightweight instrumentation at the machine IR level. MIP is specially designed to collect profile data used to effectively optimize for the metrics we care about in the mobile space while also achieving low size and runtime overhead of instrumenting binaries. Since MIP produces small instrumenting binaries, we are able to run and collect profiles in a production environment with little noticeable impact to performance. We have developed an automatic pipeline that generates and commits optimization profiles to our source code repository by regularly running MIP instrumenting builds on real devices to collect profile data.

In MIP instrumenting builds, we inject probe instructions in machine IR rather than LLVM IR instrumentation (IRPGO) which injects them at the LLVM IR level. This is a tradeoff we chose based on the optimizations we use the profiles for. Function ordering and machine outlining are late optimization passes that run close to machine IR where our profiles are collected. This is in contrast to IRPGO which instruments LLVM IR to aid function inlining early in optimization. We record instrumented machine function and machine basic block addresses in our profiles so that if debug information is available we can infer source-level profiles from our machine profiles.

MIP has several different modes for which profile data is collected to trade off profile quality for instrumentation overhead. Those modes are: function entry timestamps, function entry call counts, basic block coverage, and return address sampling from callees. **Function entry timestamps** provide information about which functions are on the startup path of the binary. When a function is called for the first time, we record and increment a global timestamp which is used to order functions by their initial call time.

Function entry call counts simply record how many times a function is called. This statistic is also collected in prior instrumentations such as IRPGO to provide insight on which functions are critical for performance.

MIP does not collect execution counts on the full control flow graph (CFG). Instead, MIP collects **boolean coverage data for each basic block**. As described in Section 1, our

binaries tend to have simple CFGs and are IO-bound so the usual block optimizations based on execution counts are unnecessary. In the mobile space, we use block coverage to aid outlining decisions, e.g., uncovered blocks will be outlined to reduce binary size.

The last mode is **return address sampling** which adds probes to callees to record their call-sites. This is different from value profiling in IRPGO which probes call-sites to record their callees, but the resulting profile is similar. At each function entry, we save the current return address value (the LR register on AArch64 [1]) to a static global buffer whose size is specified by a compiler flag. To avoid excess memory usage, we only collect a random sample of these values. We use this buffer of return address values to infer edge frequencies between call-sites and callees and construct a *dynamic call graph*, that is a call graph that includes dynamically dispatched calls. Functions like Objective-C's *objc_msgSend* [13] compute callees at runtime and cannot be tracked by value profiling in IRPGO, but they are tracked by MIP. We use the dynamic call graph to infer relative execution frequencies of basic blocks that contain call-sites using the execution frequencies of their callees. It can also be used to improve function order for runtime performance [22] by ensuring functions that are frequently called together are spatially close in the binary.

The low size overhead of MIP's instrumenting binaries is mainly due to its extractable static metadata. We must track some metadata for the functions we instrument such as function names, function pointers, and profile data pointers, but they are not needed at runtime. The compiler emits this metadata to object files that are linked normally, but the metadata is fully extracted from the final binary. We took special care to ensure extraction works on both ELF and Mach-O binary formats by using PC-relative relocations to encode function and probe data addresses and by avoiding dynamic relocations so those values are computed at build-time. Since MIP dumps unstructured raw profile data from instrumenting builds at runtime, we need the extracted metadata to read and produce an optimization profile for the compiler to consume.

At our company, we have an automated process to continuously generate updated profiles and consume them in our production builds, shown in Figure 1. We continuously build instrumenting builds that, when run on devices, automatically upload raw profile data to our servers when the app enters the background. We have a rack of devices that continuously exercise the instrumenting build to produce enough raw profiles to provide good coverage. At regular intervals, those raw profiles are post-processed to produce a single optimization profile that is committed to our repository to be consumed by the compiler for our production builds.

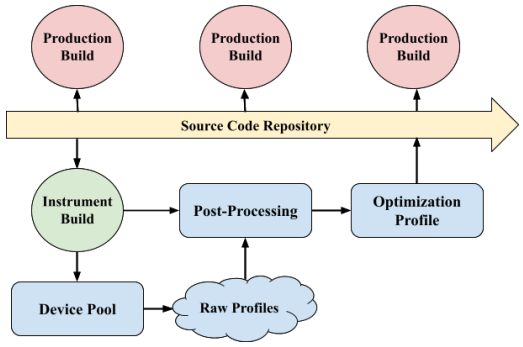

Figure 1. Overview of profile generation and consumption.

3 Outliner

Outlining reduces code size by replacing repeated sequences of instructions with function calls. Similar to outlining techniques [15, 27–29] at the LLVM IR level, LLVM also features a machine outliner [24] towards the end of compilation. As discussed in Chabbi et al. [3], outlining at the machine IR level in LLVM saves code size more effectively in mobile apps because straight-line short code sequences are heavily repetitive, which are often exposed only after the high-level IR is lowered. Yet, it leaves a lot of outlining potentials since the LLVM machine outliner runs within a single module and only finds syntactically matched sequences. This section will describe our novel approaches with three additional outliners at the machine IR level – a global outliner [19] in Section 3.1, a frame outliner [18] in Section 3.2, and a custom outliner in Section 3.3.

3.1 Global Outliner

The LLVM machine outliner operates on a single module at a time. When optimizing with regular (*full*) link-time optimization (LTO), all modules are merged into a single module, and thus the machine outliner effectively operates at a global level. However, when using the more scalable ThinLTO [16] approach, repeated instruction sequences that appear in different modules may not get outlined due to the local scope of ThinLTO.

We develop a new approach to address these shortcomings; we run machine-level code generation (but not IR-level optimizations) twice. The purpose of the first time is purely to gather statistics on outlining opportunities. The results of all individual modules are combined into a global summary. The combined knowledge is applied during machine outlining the second time.

During the first run, for each beneficial outlined instruction sequence within a module, we compute a stable hash sequence whose value is meaningful across modules. The stable hash sequence is combined with each module compilation. At the second round, the combined set of stable hashes is queried to determine whether an instruction sequence

```
OUTLINED_FUNC1:
mov  x0,x1      // Y
lsl  x0,x0,#7   // B
add  x0,x8,#77  // U
ret
```

```
OUTLINED_FUNC2:
mov  x0,x1      // Y
lsl  x0,x0,#7   // B
add  x0,x8,#33  // Q
ret
```

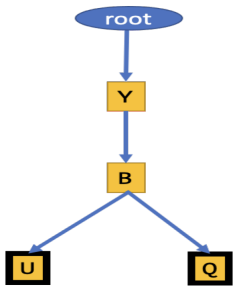

Figure 2. For each instruction of functions outlined within a module, a stable hash `Y`, `B`, `U`, or `Q` is derived. A path labeled with hashes is added to the prefix tree for each outlined instruction sequence, ending in a terminal node. Later, during the second run, other instruction sequences can be efficiently matched against the tree.

should be optimistically outlined even if the outlining would not be beneficial within the particular module. To guarantee the correctness of this optimistic approach using stable hashes, each outlined function is uniquely named so that outlined functions with identical instruction sequences will have different names. To address this duplication, we enable the link-once One Definition Rule (ODR) to let the linker deduplicate those outlined functions.

When computing stable hashes, we do not incorporate hashes of pointers, but instead expand data structures before hashing. In this way, the hashes are independent of the non-deterministic placements of data structures in memory, making hashes meaningful across modules, enabling hash comparisons even in the face of a distributed execution model, and making the hashes serializable.

To enable an efficient representation of large numbers of instruction sequence hashes, and to enable efficient queries, we introduce a new data structure: a *global prefix tree* of stable instruction hashes; see Figure 2 for an illustration. For every beneficially outlined instruction sequence during the first run, there is a corresponding path through the tree. Nodes along the path are marked with the respective instruction hashes and the final node is marked as terminal. (This is different from the suffix tree that the machine outliner [24] normally uses to determine instruction sequences that occur at least twice.) This tree is built during the first run, inserting a path of instruction hashes for each sequence that is beneficial within a module. The tree is then queried during the second run.

Algorithm 1 iterates over all instruction sequences and computes their hashes. If a sequence of hashes makes a path on the *global prefix tree* from root to terminal node, then we know there could be another module that outlines the corresponding instruction sequence and so it optimistically becomes a global outlining candidate. While the theoretical

Algorithm 1 Get a vector *Result* of instruction sequences (*StartIndex*, *EndIndex*) in the *global prefix tree.*

```
Result ← []
for I = 0 ... Size − 1 do
    Node ← Tree.Root
    for J = I ... Size − 1 do
        Inst ← get_inst(J)
        Hash ← stable_hash(Inst)
        Node ← get_next(Hash, Node)
        if Node == null then
            break
        else if Node is Terminal then
            Result.push({I, J})
        end if
    end for
end for
```

runtime of the algorithm is $O(n^2)$, in practice it is much more efficient. In our experiments with `SocialApp`, each instruction needs on average 2.5 comparisons to determine if it starts an outlined instruction sequence. Because of this, the average runtime is nearly linear in the number of instructions per ThinLTO module.

Fortunately, the time spent in code generation does not tend to dominate the overall build time, and so our approach to run code generation twice incurs an acceptable cost in practice, which will be presented in Section 5.4.

3.2 Frame Outliner

```
stp  x22,x11,[sp,#-48]!
stp  x20,x19,[sp,#16]
stp  x29,lr,[sp,#32]
```

Listing 1. Original Prolog. (16 K instances in 1 M instructions for `SocialApp`).

```
stp  x29,lr,[sp,#-16]!
bl   OUTLINED_PROLOG
```

```
OUTLINED_PROLOG:
stp  x22,x11,[sp,#-32]!
stp  x20,x19,[sp,#16]
ret
```

Listing 2. Outlined Prolog

```
ldp  x29,lr,[sp,#32]
ldp  x20,x19,[sp,#16]
ldp  x22,x11,[sp],#48
```

Listing 3. Original Epilog. (15 K instances in 1 M instructions for `SocialApp`).

```
bl   OUTLINED_EPILOG
```

```
OUTLINED_EPILOG:
mov  x16,lr
ldp  x29,lr,[sp,#32]
ldp  x20,x19,[sp,#16]
ldp  x22,x11,[sp],#48
ret  x16
```

Listing 4. Outlined Epilog

Our frame outliner runs during frame lowering, eagerly injecting calls to the outlined functions that set up or destroy the stack frame. Functions often start with a prologue that saves callee-saved registers (CSR) – e.g, `x19` thru `x29` in AArch64 [1] – into the stack frame. Likewise, functions often end with an epilogue that restores CSR from the stack frame. The current frame lowering optimizes these code sequences by combining stack adjustments, often resulting in irregular

sequences of memory operations that are not fully outlined by the LLVM machine outliner. Furthermore, any instruction operating with the link register (LR or x30) is conservatively not outlined due to its special semantic that stores the return address on a function call.

Listing 1 - 4 show how the frame outliner outlines a pair of prologue and epilogue, respectively. Unlike the prologue which needs to store the LR value at the prologue call-site, the epilogue can be completely outlined, even restoring LR by using the scratch register x16 to stash the current return address. Note the stack unwind code (not shown in the above Listings) is still in place at each prologue call-site to preserve debuggability.

3.3 Custom Outliner

Objective-C and Swift use many helper calls to support ARC [12, 33]. These helper calls often have highly repetitive instructions for setting up and passing arguments. However, they are often shuffled (or scheduled) with different allocated registers causing irregular sequences of instructions that are not outlined by the LLVM machine outliner.

Our custom outliner matches these semantically equivalent sequences of instructions and eagerly replaces them with function calls. This outliner pass is placed before the machine outliner, making the remaining instructions more regular and thus more efficiently outlined by the machine outliner that follows.

We built a simple register tracker within each basic block to match the code sequences specified by patterns. The patterns are created based on our binary inspection and registered to the outliner. We do this approach because our outlining is highly customized to take advantage of AArch64 machine instructions, which is hard to generalize. Adding new patterns to the outliner is fairly straightforward by specifying register dependencies between instructions and their lowering to the outlined function.

```
1 or   x0,xzr,x19
2 ...
3 bl   objc_release
```

Listing 5. Original Release. (42 K instances in 1 M instructions for SocialApp).

```
1 bl   OUTLINED_RELEASE
```

```
1 OUTLINED_RELEASE:
2 or   x0,xzr,x19
3 b    objc_release
```

Listing 6. Outlined Release

Listing 5 and 6 show a pattern for release operations which has dependency on x0 (the first argument) ending with *objc_release*. This simple pattern can be also exploited by the existing machine outliner but only when the instructions are consecutive in order. Rather, our custom outliner detects this sequence even if the argument set-up and the release call are scheduled and thus apart.

```
1 bl   objc_msgSend
2 mov  x29,x29
3 ...
4 bl   objc_retAutoRelRet
```

Listing 7. Original Marker. (25 K instances in 1 M instructions for SocialApp).

```
1 bl   OUTLINED_MARKER
```

```
1 OUTLINED_MARKER:
2 stp  x29,lr,[sp,#-16]!
3 mov  x29,sp
4 bl   objc_msgSend
5 mov  x29,x29
6 bl   objc_retAutoRelRet
7 ldp  x29,lr,[sp],#16
8 ret
```

Listing 8. Outlined Marker

In ARC, there is a common pattern where the caller and callee cooperate to keep the returned object out of the autorelease pool [31]. An *objc_autoreleaseReturnValue* at the callee examines a marker (e.g., `mov x29,x29` in AArch64) immediately following the call-site, which hints that an *objc_retainAutoreleasedReturnValue* will follow. If so, using thread-local storage, these calls cooperate to effectively skip a pair of releases at the callee and retain at the caller.

Listing 7 and 8 show a pattern for an *objc_msgSend* call-site. The custom outliner checks the dependency of the return value (x0) for *objc_msgSend* to the first argument of *objc_retainAutoreleasedReturnValue* (as a shorten form, *objc_retAutoRelRet* in the Listings). Note the outlined code purposely creates a stack frame to preserve the original debugging behavior, which is critical in production for crash symbolication. The additional outlined function frame can be easily muted in our visualization tool as needed.

```
1 adrp x8,SelRef
2 ldr  x1,[x8,SelRef]
3 ...
4 bl   objc_msgSend
```

Listing 9. Original SelRef. (19 K instances in 1 M instructions for SocialApp).

```
1 bl   OUTLINED_SELREF
2 .word SelRef-SecStart
```

```
1 OUTLINED_SELREF:
2 adrp x8,SecStart
3 add  x8,x8,SecStart
4 ldr  x1,[lr],#4
5 ldr  x1,[x8,x1]
6 b    objc_msgSend
```

Listing 10. Outlined SelRef

Accessing a global variable using AArch64 instructions needs at least two instructions by computing a page address (e.g., adrp) followed by a load or store with the page offset. *objc_msgSend* is the most frequently used call in Objective-C. The second argument (x1 in AArch64) points to the selector's name [13] while the user arguments follow it.

Listing 9 and 10 show an outlining sequence that sets x1 and calls *objc_msgSend*. The outliner encodes the 4 byte offset relative to the start of *__objc_selrefs* section immediately following the function call. The outlined function computes x1 by reading the custom offset at the call-site and then tail-calls *objc_msgSend*. Note the link register (LR) value is increased by 4 byte (1 instruction) so that the encoded custom offset can be skipped when returning. Similar

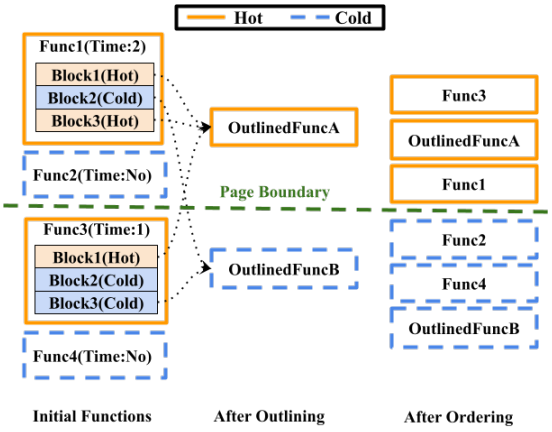

Figure 3. Hot function ordering with MIP profile data which has function timestamp and block coverage. After ordering, all hot code is grouped into a single page.

logic can be applied to any other global access like class references in the first argument of a static method call.

4 Optimization

Section 4.1 first shows how we order hot functions to minimize page faults. Then Section 4.2 covers how to control the level of size optimizations to maintain key performance scenarios. Lastly, Section 4.3 presents our cold function ordering to optimize the compressed binary size.

4.1 Hot Function Ordering

Function and block ordering [21, 22] for server workloads has been heavily studied, which mainly optimize TLB or cache performance during steady state. In mobile apps, start-up performance is critical, which is impacted by page faults [6].

Figure 3 illustrates how we order hot functions with profiles collected using MIP described in Section 2. Initially, functions are laid out in source order and may span multiple pages. If *Func*3 and *Func*1 are executed in order, MIP records the timestamps, $Func3(Time : 1)$ and $Func1(Time : 2)$, in the profile. Also, block coverage info is collected and covered blocks are colored in orange. We can infer function coverage for outlined functions using the coverage of the blocks that call them. Note outlining happens only within a block in LLVM. The same approach can be applied to functions without instrumentation like those written in assembly or in 3rd party libraries. The covered outlined function, *OutlinedFuncA*, is now placed next to its caller, *Func*3, in a greedy way.

Using the dynamic call graph described in Section 2, we have tried a few different ordering policies for non start-up functions used for server workloads. However, the impact was hardly measurable because the mobile apps we tested do not have long-running scenarios while they do have many IO operations. Nonetheless, it may be worth studying function ordering for mobile AR/VR and AI workloads that may be computationally bound.

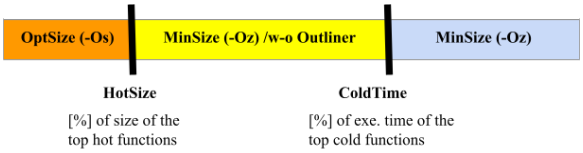

Figure 4. Functions are sorted by the estimated execution time in decreasing order. Two parameters, *HotSize* and *ColdTime*, control the optimization goal.

4.2 Optimization Goal

Although mobile apps typically do not have long-running scenarios that are computationally bound, there is a set of performance criteria related to user experience. For instance, `ChatApp` cares about latency in chat thread navigation and handling chat messages. Aggressively outlining under *-Oz* optimization may hurt these important scenarios.

To address this performance issue, we rank functions in hotness order and apply different sets of size-optimizations and control outlining. Using return address sampling (described in Section 2), we can infer the execution frequencies of call-sites statistically. The block execution frequencies can be computed from the maximum of these call-site frequencies within the block. The total execution time of a function is computed by accumulating the estimated block frequency times the block instruction size.

We sort the functions based on their total execution time estimated above in decreasing order. As shown in Figure 4, we define two knobs, *HotSize* and ColdTime: (i) *HotSize* determines the percentage of the top hot functions' size, which will be compiled with *OptSize* (*-Os*) rather than *MinSize* (*-Oz*). Intuitively, this considers the size budget for speed optimizations. (ii) *ColdTime* determines the percentage of the top cold functions' execution time, which will gate outliners. Intuitively, this considers the execution time budget for size optimizations with aggressive outlining. Note these two are independent controls and their order can be different.

Figure 5 plots total execution time in percentile as a function of total size when the functions are ranked in the hot order as described above, for `ChatApp`. Empirically, we chose the 5% *HotSize* and 5% *ColdTime* cutoffs to address any performance issues `ChatApp` cares about with only 0.5% size cost – Section 5.3 and 5.4 will have details.

4.3 Cold Function Ordering

While PGO has been traditionally used to optimize hot code, cold code was largely ignored. For mobile apps, not only the size of the binary on disk matters, but also the size of the compressed app package distributed by app stores. The amount

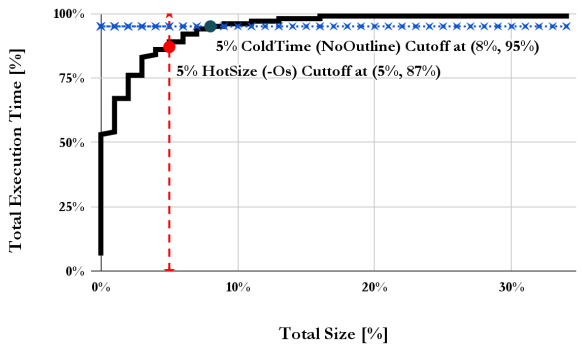

Figure 5. Total (estimated) execution time in percentile as a function of total size when the functions are ordered by execution time in `ChatApp`. The red (dashed) line shows a 5% *HotSize* cutoff which covers 87% of total execution time. The blue (crossed) line shows 5% *ColdTime* cutoff at 95% of total execution time which falls into 8% of total size.

of bandwidth used by the initial distribution of the app, and possibly weekly updates, is defined by this compressed size.

In a way similar to how outlining identifies repeated patterns in the application binary, common compression algorithms such as `ZIP` [30] or `LZFSE` [11] also identify repeating sequences and represent them in more compact ways. While the machine outliner has to operate at the granularity of whole instructions and has to respect calling conventions, general compression algorithms typically find patterns at the byte-level, and they have more freedom to efficiently represent repetitions. They typically substitute repeating sequences with pointers to the previous occurrence. The shorter the distances for those repeating sequences, the higher the compression ratio will be.

We developed a function orderer that orders cold functions based on similarity in a way that allows compression algorithms to take advantage of repeating patterns even if they could not be leveraged by the machine outliner. A similar idea has been already explored before in Redex, a bytecode optimizer [7].

Similar to the stable hashes that we introduced in the context of the global outliner, we again use stable hashes to encode machine instructions. Unlike the global outliner that considers the outlining hash sequence, this time each function is simply mapped to a set of unique stable hashes – we do not consider the order of stable hashes. By default, functions happen to be ordered by their indices in source order. Our approach implements an eager algorithm as follows, we select cold functions incrementally to produce a similarity order.

We start with the cold function with the smallest index. In every incremental step, we select the next remaining cold function with the highest similarity score, computed by crediting the number of shared hashes while penalizing that of missing hashes and that of additional hashes against the prior function. Finally, the similarity order for cold functions is concatenated with the order for hot functions found in Section 4.1. These function orders are passed to the linker that enforces them.

In this way, we achieve an overall smaller compressed binary for faster downloads while retaining faster startup and higher performant application behavior.

5 Evaluation

Section 5.1 first introduces our benchmarks and environments. Section 5.2 shows our instrumentation overhead compared to IRPGO. Then Section 5.3 presents the size and performance for mobile apps as we incrementally add our outliners and PGOs. Lastly, Section 5.4 discusses trade-offs PGO can make.

5.1 Benchmarks

We evaluated our approach on two major iOS apps being developed at our company and an open-source compiler `Clang`, using the 12.0 release; see Table 2. `SocialApp` is one of the largest non-gaming mobile apps with dozens of dynamically loaded libraries (dylibs) whose total size, including assets, is over 250 MB. This app is written using a mix of Objective-C and Swift. We built this app with ThinLTO because of build scalability concerns. `ChatApp` is a medium sized mobile app with a mix of Objective-C and C++ whose total size is over 50 MB. This app was built with LTO to maximize performance at the cost of the build time. `Clang` targets AArch64 like the other two mobile apps. In particular, we built the libLTO dylib with ThinLTO, called from the linker (LD), that natively builds another libLTO with ThinLTO as a benchmark. In this Section 5, the baseline was built with *-Oz* unless specified otherwise. The saving or reduction value in Figure or Table is calculated by 1 minus the ratio of each case over the baseline. The higher is the better and negative values indicate a regression.

Table 2. Statistics of Applications used for Evaluation.

App	Total Func.	Prof. Func.	Language	LTO Mode	OS
`SocialApp`	866 K	174 K	Obj-C/Swift	Thin	iOS
`ChatApp`	202 K	31 K	Obj-C/C++	Full	iOS
`Clang`	42 K	10 K	C/C++	Thin	MacOS

The benchmarks were cross-built on a MacOS with 18-Core Intel Xeon 2.3 GHz having 128 GB memory. Experiments for the two iOS apps were performed on iPhone8. `Clang` was experimented on Apple M1.

Table 3. Instrumenting binary size overhead. IRPGO is built with *-fprofile-generate* [32]. The overhead is calculated by the ratio of each case over the baseline size minus 1.

App	IRPGO	MIP
`SocialApp`	72.4%	21.2%
`ChatApp`	109.4%	35.5%
`Clang`	117.6%	41.5%

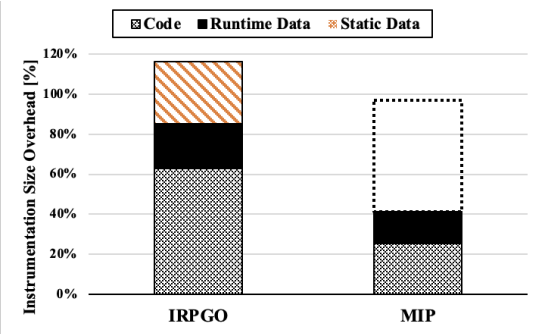

Figure 6. Instrumented binary size overhead breakdown for `Clang`. MIP extracts static data (metadata) for profile correlation into a separate map file whose portion is shown in the dotted box.

5.2 Instrumentation Overhead

Table 3 compares the (uncompressed) instrumentation size overhead of IRPGO and MIP. In this measurement, MIP uses the full instrumentation, described in Section 2, including the **return address sampling** with 2 MB buffer size. Overall MIP's overhead is only 1/3 of IRPGO's.

Figure 6 plots the breakdown of the instrumentation size overhead for `Clang`. IRPGO injects 8 byte counters for each control-flow edge, which impacts the high overhead in code compared to MIP that injects a byte coverage per block. IRPGO reserves space for runtime data like edge profile counters or value profile counters while MIP allocates space for function timestamp, function counter, block coverage, and return address sampling. The amount of code or runtime data can be closer depending on what options are chosen for IRPGO. While IRPGO emits static data like the structural information for the runtime data, and names, etc. into the final binary, MIP completely extracts our static map data into a file at build time, which makes a huge difference in the instrumentation size overhead.

5.3 PGO Size and Performance

This section compares improvements in compressed and uncompressed size, and performance for `SocialApp`, `ChatApp` and `Clang`. The baseline disables all outliners. In the comparison, all the outliners are incrementally added in order – **default outliner** (LLVM machine outliner [24]), **global outliner**, **frame outliner**, and **custom outliner**, described in Section 3. The PGOs described in Section 4 – **hot function order**, **optimization goal** for `ChatApp` and `Clang`, and **cold function order** – are then added in order. The uncompressed size is total binary size including assets. The compressed size is measured after `ZIP` [30] compression is applied. Start-up performance is measured in page faults for mobile apps. Other performance metrics that are CPU-bound will be evaluated in `ChatApp` and `Clang`.

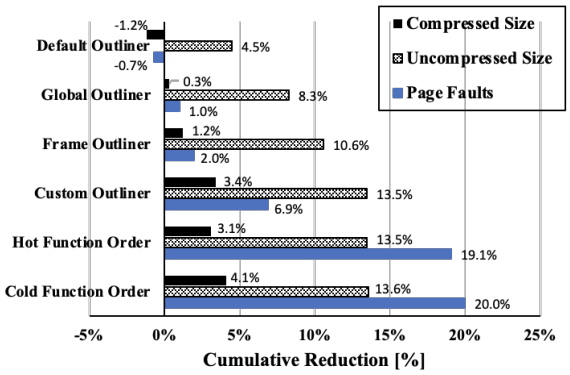

Figure 7. `SocialApp`: Cumulative reduction in compressed and uncompressed size, and major page faults.

Figure 7 shows reductions in size and page faults for `SocialApp`. The **default outliner** reduced the uncompressed size by 4.5% while it actually increased the compressed size by 1.2%. This happens because factoring the common code out of the function body might increase unpredictability of code sequence which negatively impacts the compression rate. Page faults are a bit regressed by 0.7% because the outlined functions might span across page boundaries. As our aggressive outliners are enabled, both size and start-up performance are improved because of the large code size reduction. The **hot function order** noticeably improved the page faults reduction from 6.9% to 19.1% by compacting start-up functions into fewer pages. Finally, the **cold function order** reduced the compressed size by 1% further without hurting performance.

Figure 8 shows improvements in size and performance for `ChatApp`. Because `ChatApp` is built with LTO which already has a global view, the **default outliner** and the **global outliner** are the same. Similar to `SocialApp` shown in Figure 7, the **frame outliner** and the **custom outliner** improved the uncompressed size saving further by 2 – 3% for each. The **hot function order** significantly improved the page faults reduction from 12.8% to 41.9%.

`ChatApp` tightly interacts with users in real-time, and so *thread navigation latency* is a critical performance metric for user experience. This metric is computationally expensive, and so the **hot function order** does not address the regression from aggressive outlining. As shown in Figure 5

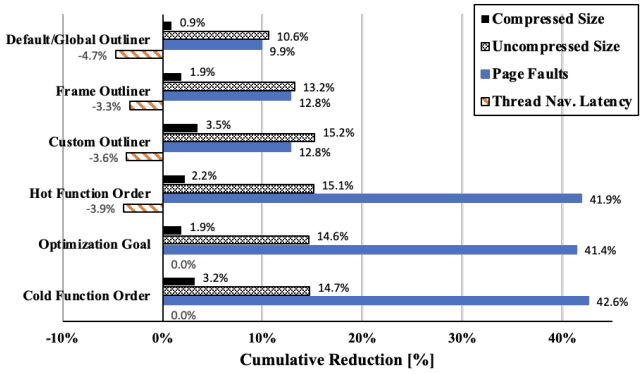

Figure 8. `ChatApp`: Cumulative reduction in compressed and uncompressed size, major page faults, and thread navigation latency.

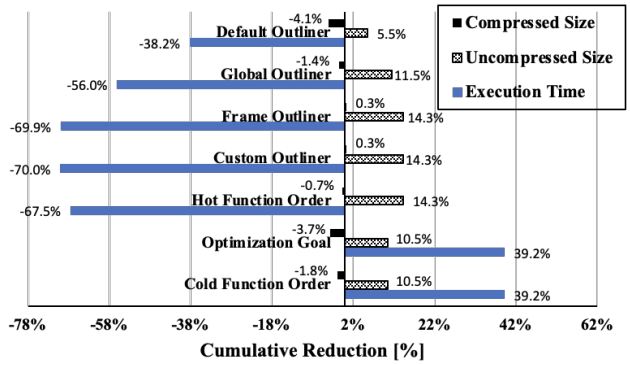

Figure 9. `Clang`: Cumulative reduction in compressed and uncompressed size, and execution time.

in Section 4.2, we optimized app performance and size by varying *HotSize* cutoff between 2 − 10% and *ColdTime* cutoff between 2 − 10%. Empirically, we chose the 5% *HotSize* cutoff and the 5% *ColdTime* cutoff. This **optimization goal** completely removed the performance regression in *thread navigation latency* from 3.9% down to 0% at the cost of only a 0.5% uncompressed size regression (from 15.1% to 14.6%). Lastly, the **cold function order** reduced the compressed size further from 1.9% to 3.2%.

Figure 9 shows reduction in the compressed and uncompressed size, and total execution time of `Clang`. Similar to `SocialApp`, `Clang` is built with ThinLTO, and the **global outliner** doubled the uncompressed size reduction from 5.5% to 11.5%. The **frame outliner** improved the uncompressed size saving further by around 3%. However, the **custom outliner** is not effective for `Clang` written in C/C++ because the patterns implemented in the **custom outliner** mainly target Objective-C or Swift.

Unlike the prior two mobile apps, `Clang` is computationally intensive. Any aggressive outlining significantly regressed the total execution time up to around 70%. The **hot function order** made a small dent by only 3% in steady-state performance because it largely impacted start-up performance. However, the **optimization goal** completely reversed the performance direction from a regression of 67.5% to an improvement of 39.2%. In this measurement, we used the same 5% *HotSize* and 5% *ColdTime* cutoffs as those of `ChatApp`. Once again, the **cold function order** improved the compressed size from a regression of 3.7% to a regression of 1.8%. Interestingly, outlining generally does not improve the compressed size in this C/C++ benchmark.

5.4 Trade-Offs

This section first shows the build-time trade-off that our global outliner made. Then we present the similarity ordering's impact on the compressed size with different compressors. Lastly, we compare our PGO driven from MIP against LLVM PGO driven IRPGO when optimizing `Clang`.

Table 4. Global outliner's impact on uncompressed size saving and build time slowdown for `Clang`. The baseline disables outliners with ThinLTO. The slowdown is calculated by the ratio of each case to the baseline build time.

	Uncompressed Size Saving	Build Time Slowdown
ThinLTO (**Default Outliner**)	5.5%	1.05X
ThinLTO (**Global Outliner**)	11.5%	1.41X
LTO (**No Outliner**)	3.1%	2.58X
LTO (**Default/Global Outliner**)	12.0%	2.84X

Table 4 compares the uncompressed size saving and the build time slowdown when enabling the global outliner for `Clang`, described in Section 3.1. Enabling the global outliner with ThinLTO doubles the size saving from 5.5% to 11.5%, which is close to the default outliner with LTO, 12.0%. The total compilation time of ThinLTO with the global outliner was increased from 1.05*X* to 1.41*X* by repeating code generation, but it is merely a fraction of the total LTO build time. For a large app like `SociaApp`, LTO is not an option in practice, and the global outliner with ThinLTO is more impactful to save both build time and code size.

Table 5. Similarity orderer's impact on compressed size and compression rate when compressing `Clang` with different compressors. The compressor rate is calculated by the ratio of the uncompressed size to the compressed size.

	ZIP	LZFSE	xz32	xz
Compressed Size Saving	5.9%	4.5%	5.7%	1.0%
Compressor Rate	2.47X	2.45X	3.10X	3.43X

Table 5 shows the compressed size saving with our similarity ordering, described in Section 4.3, and the compression multiplier when compressing `Clang` with different compressors. For this measurement, all functions are ordered based on similarity to show the upper bound of the compressed size win. `ZIP` [30] compression can save up to 5.9% compressed size. `LZFSE` [11] is the one Apple App Store uses, showing 4.5% saving with the default setting. `xz32` is a variant of `xz` [8], which we tested with a set of parameters, *-9fk –lzma2=dict=32k*. Comparing `xz32` with `xz`, the compressor multipliers are higher than those for `ZIP` or `LZFSE`, but the size savings are drastically different, 5.7% and 1.0% respectively. This means the effectiveness of our similarity ordering doesn't depend on the compressor multiplier, but instead on how close similar functions are and the directory size that compressors can hold. It may be worthwhile to tune the similarity orderer based on the compressor used.

Table 6. Size reduction in the compressed and uncompressed size, and speed-up using the default LLVM PGO, our MIP PGO, Chabbi et al. [3], and *-O3* for `Clang`. The speed-up is calculated by the ratio of the baseline execution time to each case.

	Compressed Size Saving	Uncompressed Size Saving	Speed-up
LLVM PGO (*-Oz*)	-11.8%	-7.1%	2.28X
MIP PGO (*-Oz*)	0.6%	3.7%	2.53X
Chabbi (*-Oz*)	0.0%	0.7%	0.98X
Baseline (*-O3*)	-22.7%	-26.2%	4.52X

Similar to `ChatApp`, we exhaustively evaluated `Clang`'s performance and size by varying *HotSize* between 2 – 10% and *ColdTime* between 2 – 10%. `Clang` is CPU-bound and sensitive to the optimization settings described in Section 4.2. These two cutoffs (*HotSize*, *ColdTime*) affected both the performance and the size proportionally in opposite directions. For instance, if we tune it towards size, like (2%, 10%), the reductions in the uncompressed size and the total execution times are 11.8% and 29.2%, respectively. On the other hand, if we tune it toward speed, like (10%, 2%), the reductions are 8.9% and 45.3%, respectively. Note any of these combinations produced a large win, but the sizes of the wins were shifted.

Table 6 compares the size reduction and speed-up in the default LLVM PGO (*-fprofile-use* [32]) driven from LLVM IR instrumentation (*-fprofile-generate* [32]) vs. our PGO driven from MIP. Naively applying LLVM PGO actually regressed both compressed and uncompressed size by 11.8% and 7.1% respectively, although it improved performance by 2.28*X*. Our PGO pipeline using MIP could improve both the compressed and uncompressed size, and improve speed-up by 0.6%, 3.7% and 2.53*X*, respectively, when using the 10% *HotSize* cutoff and 2% *ColdTime* cutoff found above. In short, when directly compared to LLVM PGO, our PGO produced roughly a 10% smaller and 10% faster `Clang` binary with only 1/3 of the instrumentation overhead compared to IRPGO. This shows how efficiently our PGO framework can improve both performance and size for workloads that are constrained by size.

We also compared MIP PGO against Chabbi et al. [3] and *-O3* in Table 6. Chabbi et al. [3] proposed code size saving techniques at the machine IR level for commercial iOS apps, similar to ours. They merged all modules to get the whole-program view to apply outlining techniques while our global outliner at ThinLTO scales well for large mobile apps. Note our baseline is *-Oz* which already includes the default machine outliner. We also measured the entire binary size instead of just code size. The additional size savings from repeating the outliner pass up to 5 times in the Chabbi's work was less than 1% while it regressed speed-up due to additional call-overhead from outlining without considering profile data. Expectedly, *-O3* greatly improved performance at the cost of large size regressions.

6 Related Work

Order file instrumentations [26] and function entry instrumentations [17] are lightweight instrumentations at the function level. Phipps et al. [25] proposed an instrumentation that strips static metadata, similar to MIP. Like the above two prior works, it is confined to a particular use case, code coverage, while MIP can be used with various modes including PGO.

Outlining techniques [15, 27–29] reduced code size at the LLVM IR level, but repeated sequences are exposed heavily at the machine IR level for iOS apps we target. They operated with the entire IR to find outlining opportunities, which does not apply to ThinTLO we use for large apps. Our outliners scale well by optimistically outlining the candidates per module while the linker deduplicates them.

Binary code similarity analysis is an established field [9], typically done for security and privacy analysis and reverse engineering, while we employ it in the back-end of the compiler to improve compression. Baker et al. [2] and Motta et al. [20] find similarities to enable efficient representations of program patches, while we are reducing the compressed size of a single application.

To the best of our knowledge, no other prior work evaluates how much aggressive outlining will hurt performance for real-world mobile apps when they must be optimized for size, and what trade-offs we can make using relevant profile information. We comprehensively presented an end-to-end approach consisting of instrumentation, outlining, optimization, and even compression for the deployment of real-world mobile apps.

7 Conclusion

This paper described our whole PGO framework with two real-world mobile apps, which were being developed at our company. We presented a lightweight instrumentation, MIP, whose overhead is only 1/3 of IRPGO's. We proposed a global outliner, a frame outliner, and a custom outliner to improve upon the LLVM machine outliner. When combined, the uncompressed size saving was up to 3X of the machine outliner (from 4.5% to 13.5%) for `SocialApp`. Our PGO improved start-up performance by reducing page faults up to 42% while maintaining key performance metrics, and lastly squeezed the compressed binary size further by 1% for `ChatApp`. Finally we evaluated the above in the context of a CPU benchmark, `Clang`, which became 10% smaller and 10% faster compared to LLVM PGO with *-Oz* optimization.

Acknowledgments

We would like to thank Manman Ren and Minjang Kim for their help designing a previous iteration of MIP. We are greatful for the support of Chris Parker, Julien Lerouge, and many others on infrastructure and product teams at Meta. Finally, we would like to thank the anonymous reviewers for their helpful suggestions.

References

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

[2] Brenda S. Baker, Udi Manber, and Robert Muth. 1999. Compressing differences of executable code. In *ACM SIGPLAN 1999 Workshop on Compiler Support for System Software (WCSSS'99)*.

[3] 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, NY, USA, 363–366. https://doi.org/10.1109/CGO51591.2021.9370306

[4] 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

[5] Dehao Chen, Tipp Moseley, and David Xinliang Li. 2016. AutoFDO: Automatic feedback-directed optimization for warehouse-scale applications. In *2016 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 12–23.

[6] Malcolm C. Easton and Ronald Fagin. 1978. Cold-Start vs. Warm-Start Miss Ratios. *Commun. ACM* 21, 10 (Oct. 1978), 866–872. https://doi.org/10.1145/359619.359634

[7] Facebook. 2021. *ReDex: An Android Bytecode Optimizer*. https://github.com/facebook/redex/blob/master/libredex/MethodSimilarityOrderer.h

[8] GNU. 2020. XZ Utils. https://tukaani.org/xz

[9] Irfan Ul Haq and Juan Caballero. 2021. A Survey of Binary Code Similarity. *ACM Comput. Surv.* 54, 3, Article 51 (April 2021), 38 pages. https://doi.org/10.1145/3446371

[10] Ellis Hoag and Kyungwoo Lee. 2021. RFC: Machine IR Profile. https://lists.llvm.org/pipermail/llvm-dev/2021-June/151086.html

[11] Apple Inc. 2015. LZFSE compression library and command line tool. https://github.com/lzfse/lzfse

[12] Apple Inc. 2021. Automatic Reference Counting. https://docs.swift.org/swift-book/LanguageGuide/AutomaticReferenceCounting.html

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

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

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

[16] 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, USA) *(CGO '17)*. IEEE Press, 111–121.

[17] Aditya Kumar, Ian Levesque, and Sam Todd. 2020. Cheap Function Entry Instrumentation. https://llvm.org/devmtg/2020-02-23/slides/Aditya-FunctionEntryInstrumentationLLVM.pdf

[18] Kyungwoo Lee. 2020. Homogeneous Prolog and Epilog for Size Optimization. https://lists.llvm.org/pipermail/llvm-dev/2020-March/140206.html

[19] Kyungwoo Lee and Nikolai Tillmann. 2020. Improving Machine Outliner for ThinLTO. https://llvm.org/devmtg/2020-02-23/slides/Kyungwoo-GlobalMachineOutlinerForThinLTO.pdf

[20] Giovanni Motta, James Gustafson, and Samson Chen. 2007. Differential Compression of Executable Code. In *2007 Data Compression Conference (DCC'07)*. 103–112. https://doi.org/10.1109/DCC.2007.32

[21] Andy Newell and Sergey Pupyrev. 2020. Improved Basic Block Reordering. *IEEE Trans. Comput.* 69, 12 (2020), 1784–1794. https://doi.org/10.1109/TC.2020.2982888

[22] Guilherme Ottoni and Bertrand Maher. 2017. Optimizing function placement for large-scale data-center applications. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 233–244. https://doi.org/10.1109/CGO.2017.7863743

[23] Maksim Panchenko, Rafael Auler, Bill Nell, and Guilherme Ottoni. 2019. BOLT: A Practical Binary Optimizer for Data Centers and Beyond. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2–14. https://doi.org/10.1109/CGO.2019.8661201

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

[25] Alan Phipps and Cody Addison. 2020. Source-based Code Coverage for Embedded Use Cases. https://llvm.org/devmtg/2020-09/slides/PhippsAlan_EmbeddedCodeCoverage_LLVM_Conf_Talk_final.pdf

[26] Manman Ren. 2019. Order File Instrumentation. https://lists.llvm.org/pipermail/llvm-dev/2019-January/129268.html

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

[28] 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, NY, USA, 110–121. https://doi.org/10.1145/3461648.3463852

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

[30] Greg Roelofs. 2008. Info-ZIP. http://infozip.sourceforge.net/Zip.html

[31] Apple Open Source. 2021. *Source code for objc-arr.mm*. https://opensource.apple.com/source/objc4/objc4-493.9/runtime/objc-arr.mm

[32] The Clang Team. 2021. Clang Compiler User's Manual. https://clang.llvm.org/docs/UsersManual.html#profile-guided-optimization

[33] The Clang Team. 2021. Objective-C Automatic Reference Counting (ARC) — Clang 13 documentation. https://clang.llvm.org/docs/AutomaticReferenceCounting.html