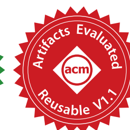


An Experience with Code-Size Optimization for Production iOS Mobile Applications

Milind Chabbi
Programming Systems Group
Uber Technologies
Palo Alto, USA
milind@uber.com

Jin Lin
Programming Systems Group
Uber Technologies
Palo Alto, USA
jinl@uber.com

Raj Barik
Programming Systems Group
Uber Technologies
San Francisco, USA
rajbarik@uber.com

***Abstract*—Modern mobile application binaries are bulky for many reasons: software and its dependencies, fast-paced addition of new features, high-level language constructs, and statically linked platform libraries. Reduced application size is critical not only for the end-user experience but also for vendor's download size limitations. Moreover, download size restrictions may impact revenues for critical businesses.**

In this paper, we highlight some of the key reasons of code-size bloat in iOS mobile applications, specifically apps written using a mix of Swift and Objective-C. Our observation reveals that machine code sequences systematically repeat throughout the app's binary. We highlight source-code patterns and high-level language constructs that lead to an increase in the code size. We propose whole-program, fine-grained *machine-code outlining* as an effective optimization to constrain the code-size growth. We evaluate the effectiveness of our new optimization pipeline on the `UberRider` iOS app used by millions of customers daily. Our optimizations reduce the code size by 23%. The impact of our optimizations on the code size grows in magnitude over time as the code evolves. For a set of performance spans defined by the app developers, the optimizations do not statistically regress production performance. We applied the same optimizations to Uber's `UberDriver` and `UberEats` apps and gained 17% and 19% size savings, respectively.

***Index Terms*—code-size, machine outlining, iOS, swift, inter-module optimization, whole-program optimization**

I. Introduction

`UberRider` is Uber's flagship mobile app used by several million active users worldwide to assist in their transportation needs. Uber's business model depends primarily on the mobile app, as is the case with many other modern businesses [1]. The fast-growing business demands rapid feature enhancements to the `UberRider` app, which has put a tremendous burden on the application's binary size. Furthermore, Apple App Store [2] imposes a limit on the app size when downloading over the data plan. Any app larger than the limit can only be downloaded over the Wi-Fi. The download limits depend on the OS version, and they have been 100MB, 150MB, and 200MB in 2017, 2019, and 2020 respectively. This critical restriction means first-time users cannot download the app when they need it the most, and the company cannot deliver features or security updates to existing users when they are not on Wi-Fi. We established a correlation between the `UberRider` app size and customer engagement — when the app size crosses the download size limit, and it leads to a 10% reduction in app installations, 12% reduction in sign-ups, and 20% reduction in first-time bookings resulting in revenue loss.

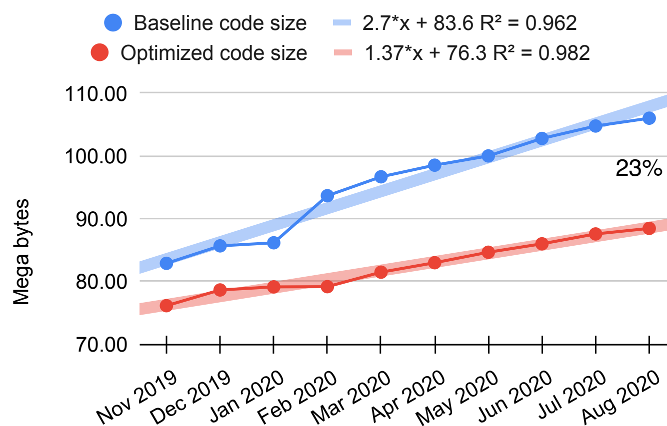


Fig. 1: The `UberRider` app's code-size grows rapidly (the blue curve with dots). The machine-code outlining optimization delivers 23% size reduction (the red curve with dots). The thick straight lines show the regression lines and the corresponding equations appear on top of the chart along with their R^2 values. The optimizations change the slope of the line and reduce the code-size growth rate by $\sim 2\times$.

During the past three years, the `UberRider` app has often reached closer to the App store download limit. Over 92% of the total app size is the application binary; the remaining size is due to media and resources. Over 77% of the binary is the machine instructions; the remaining size is due to data embedded in the binary. The blue line in Figure 1 shows the code size growth had we not applied any of the optimizations discussed in this paper. The red line shows the effects of our size-reduction optimizations: notably, first, we reduce the code size by 23%, and second but more importantly, we reduce the rate of code size growth by $\sim 2\times$.

Since the `UberRider` app was growing in code size, we set out to reduce it with the following objectives.

1) Bring the size to be well under the App Store download limit. Smaller the better.
2) Choose the optimization that continuously delivers impact for the foreseeable future as the app evolves.
3) Be transparent so that application developers are not asked to divert their energies towards size reduction.
4) Do not regress the performance of critical use cases.

Accepted for publication by IEEE. © 2021 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/ republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

5) Do not increase *local* build times since the build time is a critical developer productivity factor.

Code size is at the heart of many compiler optimizations [3], [4] such as common sub-expression elimination [5], partial redundancy elimination [6], copy propagation [7], inlining [8], value numbering [9], constant propagation [10], dead and unreachable code elimination [4], compiler-pass reordering [11]–[14], code compression [15]–[17], register allocation and instruction scheduling [18], and peephole optimizations [19]. `UberRider` is already compiled for size using the iOS build pipeline (which uses LLVM [20]), but they were insufficient to reduce our app size growth. Link-time and post-link-time optimizations [21]–[28] have shown reasonable success in code-size reductions, which we employ in our work.

We systematically assessed the `UberRider` app software at various granularities — module, file, function, basic block, and sequences of machine instruction. Our insights demonstrated several common patterns of machine code repetition. We identified outlining a sequence of machine instructions (aka machine outlining) as an effective optimization to reduce code size. Outlining [29], [30] is the opposite of inlining [8] and can hurt performance if done in hot loops. Loops hotspots, however, are uncommon in UI-intensive apps such as `UberRider`, `UberDriver`, and `UberEats`.

Machine-code outlining is available in LLVM, but a naive application of machine outlining was not beneficial; hence, we developed a compilation pipeline that could make machine outlining deliver benefits at the whole-program level. We further identified the limitations of machine outlining on how it misses opportunities and developed *repeated machine outlining* to extract more code size reduction. The result is a significant code-size reduction in `UberRider` (23%), `UberDriver` (17%), and `UberEats` (19%) apps with no statistically significant performance regression and zero involvement from our feature team developers. In the process, we encountered several hurdles and limitations arising from our multilingual app and performance regressions, which we describe in §VI.

Our optimizations are (a) in production and used by millions of Uber customers, (b) upstreamed to the open-source LLVM, and (c) not only holding up their promises build after build, but their impact on code size is growing over time. We make the following contributions in this paper.

1) Highlight common patterns of machine code repetitions and pinpoint their causes.
2) Employ whole-program machine-code outlining as an effective means to reduce code size.
3) Invent *repeated machine outlining* to gain additional size reduction, which accounts for 27% of the total size saving.
4) Mitigate performance regressions arising from our optimizations via a data layout optimization.
5) Evaluate, in detail, the impact of our techniques on the `UberRider` iOS mobile application in use worldwide by several million daily users.

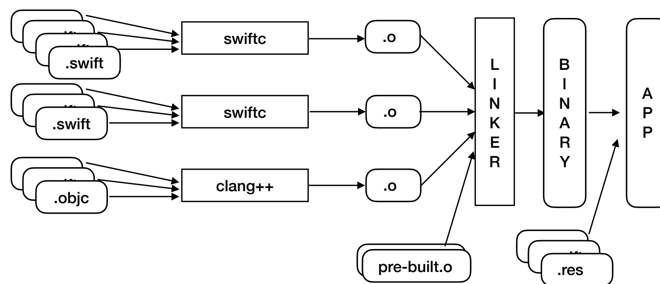


Fig. 2: The default iOS build pipeline.

6) Demonstrate 23% size reduction in the `UberRider` app and 2× reduction in code-size growth over time as a result of our optimizations.

In the rest of this paper, we use the `UberRider` app for a deep dive; the findings discussed herein are corroborated by the `UberDriver` and `UberEats` apps, and the optimizations are deployed in production, delivering similar benefits.

II. Background and iOS App Build Chain

The `UberRider` app is written in a mix of Swift and Objective-C programming languages. Swift is a statically compiled programming language, which is gaining in popularity among iOS developers. Swift offers reliability and productivity features including strong static typing, extensive error handling constructs, and automatic memory management via reference counting that makes it an attractive alternative to Objective-C. Swift is placed at 12th rank, and Objective-C is at 19th rank according to TIOBE [31] programming language ranking.

A. Default iOS Build Pipeline

Figure 2 depicts the default build pipeline used by iOS applications, including the `UberRider` app's build pipeline prior to the work described in this paper. The workflow involves compiling all the source files in a module to produce an `AArch64` [32] object file. Several such modules are independently compiled; since `UberRider` is multilingual, it also compiles Objective-C files separately into object files. All object files, including any pre-built binaries, are linked with the system linker into the final binary. The app itself may package additional resources.

The overall flow of the Swift compiler is shown in Figure 3. A Swift file is parsed to an AST on which various semantic checks are performed. Subsequently, SILGen generates Swift Intermediate Language (SIL) and performs various optimizations. IRGen lowers SIL to LLVM IR, which is optimized and compiled to an object file.

B. The `UberRider` App

The `UberRider` app has about two million lines of code, with $\sim 83\%$ in Swift and the rest in Objective-C. The source code for `UberRider` consists of 476 modules, out of which 62 are vendor-specific libraries, including RxSwift [33], SnapKit [34], Swift-NIO [35], Freddy [36], and Swift-Protobuf [37]. Individual modules are compiled using the whole-module

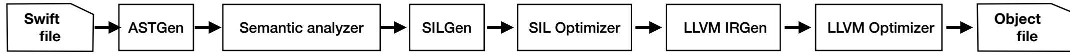


Fig. 3: Swift compiler passes.

optimization (-wmo) in the Swift compiler, which performs inter-procedural optimizations within a module. We use the -Osize flag to produce a size-optimized binary.

The UberRider app developers have employed several linting rules to guard against binary size explosion, which include avoiding large value types (e.g., struct and enum), restricting access control levels to the lowest (e.g., avoid public and open accesses when possible), avoiding excessive use of generics, and using final attributes. The app build infrastructure employs several in-house static analysis tools for removing dead-code [38] and resources and disabling reflection metadata to reduce binary size. Although these techniques together reduce the app's size, opportunities for cross-module optimizations are still left unexplored, which is one of the focuses of this paper.

C. LLVM MachineOutliner

The LLVM compiler recently introduced a MachineOutliner pass [39], which targets the code size problem. Our work builds upon this optimization.

MachineOutliner is a target-specific pass that runs late in the optimization pipeline after register allocation. It maintains machine instructions belonging to every basic block of a function in a suffix tree. For each unique sequence (representing a repetition), it creates a new outlined function (if proven safe and profitable), and then substitutes each occurrence with a call/branch to the outlined function. The order of choosing the outlining candidates is a greedy heuristic.

Outlining a sequence of machine instructions out of a function is not free: first, code patches, if necessary at the extracted location or in the new function, may eliminate the size benefit; and second, the new function call can introduce a performance penalty. LLVM heuristic ignores the performance cost, but it is cognizant of the size benefit. Each supported architecture (x86 and AArch64) supplies information on how a sequence of instructions should be outlined including estimates on the bytes needed to create an outlined function. This cost model drives the outlining choice.

The optimal solution to outlining can be cast into the knapsack optimization problem, which is NP-hard [29], [30]. MachineOutliner employs a greedy heuristic that picks the pattern from a pool of patterns that saves the most size immediately. We define a *substring* as a contiguous sequence of instructions within another instruction sequence. Assume an instruction sequence α is chosen as the first sequence to outline. If a lengthier sequence, β, has a *substring* α, then the α part of β will be outlined, but the MachineOutliner discards the rest of β from further consideration.

TABLE I: SUMMARY OF DIFFERENT OPTIMIZATION CHOICES.

Level	Optimization considered	Note
AST	Source function replicas [40]	<1% replication
SIL	SIL outlining [41]	0.41% size saving
LLVM-IR	MergeFunction [42]	0.9% size saving
	FMSA [43]	2% size savings
	MergeSimilarFuncs [44]	high build times
	IROutliner [45]	not target aware
ISA	Repeated machine outlining	23% size reduction

III. THE LANDSCAPE OF BINARY-SIZE SAVINGS

Code duplication can be detected at different levels: Swift AST, Swift Intermediate Language (SIL), LLVM IR, and machine code. Table I depicts the entire landscape and opportunities for savings that we investigated before arriving at our final cross-module machine-code level outlining choice.

AST: Static clone detection tools such as SourcerCC [46] and PMD [40] tokenize the source files and, by using a similarity measure, can identify clones at a finer granularity. Although we have deployed PMD in our production pipeline, the clone reports produced periodically contain high degrees of false-positives (this is not surprising for a static analysis tool), often leading to deprioritization compared to, say new feature development, in a fast-paced environment.

SIL: The SILOptimizer component in Figure 3 enables an "Outlining" [41] pass that creates function calls in lieu of inlined instruction sequences for certain well-defined patterns such as copy, assignment, and reference counting on value types. However, the impact of this optimization on the UberRider app is negligible — only 0.41% size saving.

LLVM IR: The MergeFunction [42] pass in LLVM merges functions with identical IR. The impact of this optimization on the UberRider app is negligible — less than 0.9% size saving. We also explored upcoming code-size optimizations in the LLVM community: Function merging by sequence alignment (FMSA) [43], [47] delivered a 2% size reduction with one-hour compilation time; MergeSimilarFuncs [44], [48] could not complete compilation in our stipulated 24-hour build time, which could be because it is very recent and not tested on large Swift+Objective-C codebases; finally, we had trouble getting the IROutliner [45] to work for all modules of UberRider.

ISA: Since a high-level IR instruction often lowers to more than one machine instruction, prior approaches cannot identify clones at the sub-IR-opcode level. Thus the best granularity of clone detection is at the machine level. When done at link or post-link time, this approach offers an ultimate window into observing the entire binary; it reveals clones introduced by prior layers irrespective of its provenance — source code, language compiler, IR, and machine-code generation. This is the approach taken by us.

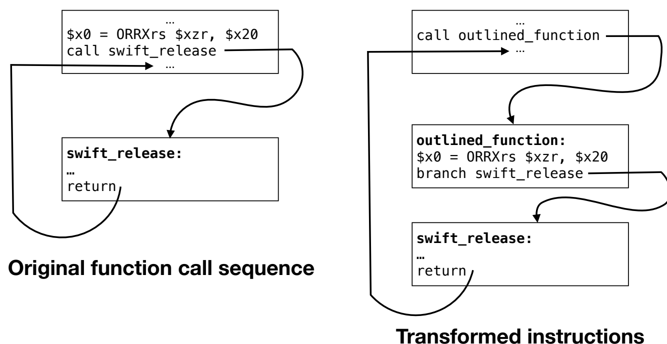


Fig. 4: An example of profitably outlining a short two instruction sequence ending with a call instruction. The outlined function exploits the tail call optimization.

IV. Machine-code Replication Patterns: A Binary Analysis

In this section, we summarize our findings by systematically analyzing the repeating machine-code sequences in the `UberRider` app. We use the term *pattern* to mean a unique instruction sequence; no two patterns are the same. We use the term *candidate* to mean an instance of a pattern; two or more candidates can match the same pattern.

Single instruction replicas are abundant, but they cannot be replaced profitably on a fixed-instruction width architecture such as `AArch64`; the cost of replacing the cloned instruction is higher than retaining the original instruction. Thus, we don't investigate single-instruction replicas. On the other hand, instruction patterns of length two or more can be profitable for outlining. That is, we can substitute them with a shorter sequence, typically a single `call` or an unconditional `branch` instruction. This requires transferring the control to an outlined instruction sequence that effectively executes the original sequence of instructions and then resumes at the instruction immediately following the original sequence. We limit our study to basic-block boundaries for two reasons: (1) the opportunity to replace lengthier patterns is small, which becomes evident later in this section, and (2) it is difficult to reason about profitability in the presence of control-flow.

Figure 4 shows an example of a profitable 2-instruction sequence ending with a `call` instruction found in the `UberRider` app. These two instructions can be replaced with a `call` instruction to a newly created outlined function; the outlined function executes the prefix instruction(s) and finally tail-calls the original function call. Similarly, a 2-instruction sequence ending with a `return` instruction can be outlined by simply introducing a jump to the outlined function; the outlined function will consist of the original 2-instruction sequence. Such patterns ending with a `call` or a `return` instruction are the most common ones; they account for 67% of all the profitable and repeating candidates in the `UberRider` app.

Since the default iOS build pipeline (described in §II-A) does not facilitate optimizations on the whole-program level, we built an alternate pipeline (shown in Figure 10, details in §V-A), which provides the ability to outline machine code sequences at the whole program-level. In this new pipeline, we introduce a statistics collection pass after machine-code generation to log the patterns of machine instructions. Our pass runs late in the `llc` phase of the LLVM compiler after all the machine-code optimizations, including register allocation, have been performed. The pass logs the patterns with their frequency of repetitions (high-to-low) including the corresponding function names and source files for further investigation.

We report only those patterns that yield at least one-byte size saving if outlined in the entire binary. The profitability includes the overhead of instructions introduced (if any) for saving and restoring registers at the call site and frame creation and destruction cost (if any) at the newly created function. By inspecting the patterns of repeated machine-code sequences meeting this profitability criterion across the entire application binary, we make the following key observations.

(1) Machine-code sequences repeat frequently, and the frequency of repetition follows the power-law curve. Figure 5 plots the frequency of repetition in machine-code sequences (blue line) overlaid with the sequence length (red line). The x-axis denotes the unique-id of each pattern where the highest occurring pattern is given an id 1, the next highest is given an id 2, and so on. It is a log-log graph. A few patterns repeat very frequently, but there is also a very long tail of patterns each progressively repeating fewer times, which obeys the *power-law* ($y = ax^b$) with 99.4% confidence. Listings 1-8 show a few most recurring patterns.

Figure 6 shows the same red line of Figure 5, however, the x-axis is not on the log scale. The red line reveals a recurring fractal pattern [49] — frequently occurring patterns have a very short sequence length (left side); as the frequency decreases, the diversity of sequence lengths increases (right side). The data points from one spike to the next spike on the x-axis represent a cluster of patterns that **repeat the same number of times**; within each cluster, there are very few lengthy sequences, but as the sequence length reduces, a larger and larger variety of patterns emerge. Finally, comparing one cluster on the left (higher repetition frequency) with another cluster on the right (lower repetition frequency), it is obvious that, as the repetition frequency decreases, both the variety of patterns (the length of horizontal steps) and sequence lengths (the height of spikes) increase.

Figure 7 plots the cumulative size savings possible by outlining the next most profitable pattern (x-axis). A lot of patterns ($> 10^5$) need to be outlined to extract most ($> 90\%$) of the possible size gain. One cannot "hard-code" a few patterns and hope to gain a significant benefit.

(2) Patterns of length only two occur most commonly, and lengthier patterns are quite infrequent This finding is evident because as the length of the pattern increases, finding a match naturally reduces. Figure 8 depicts a histogram with bins representing different sequence-lengths, and the y-axis is the number of candidates of a sequence-length found in the entire program. There are far more repetitions of shorter patterns than the lengthier ones. The longest repeating pattern is 279 instructions long and repeats three times (clipped in the

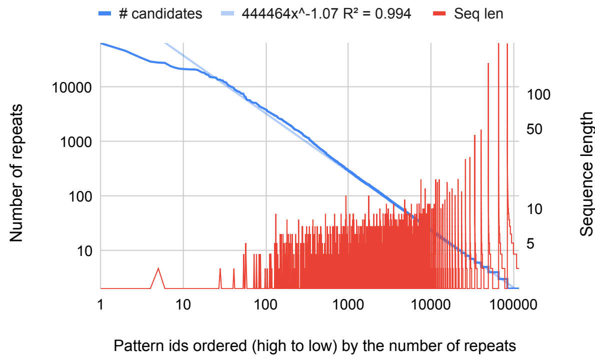


Fig. 5: Each point on the x-axis (log scale) is an instruction sequence pattern ordered (high to low) by its frequency of repeats. The number of candidates per pattern (blue line) follows a power-law curve with a long tail. The red line plots the sequence length for each pattern.

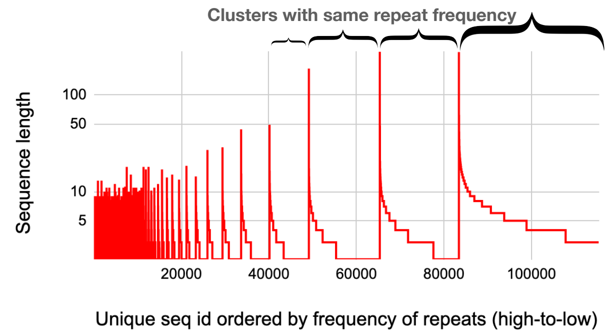


Fig. 6: The fractal nature of the sequence length of repeating instructions; at very high repeat counts, the sequences are of short length; low repeat counts offer a wide variety of sequence lengths and there are more unique sequences of shorter lengths.

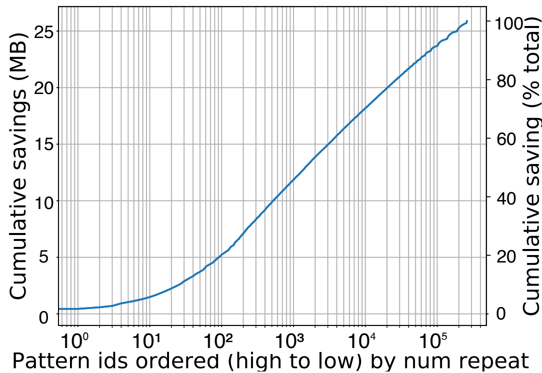


Fig. 7: The x-axis is sorted by the patterns with highest to lowest repeat frequency, y-axis is the cumulative savings possible. Numerous ($> 10^5$) patterns should be outlined to gain most of the savings.

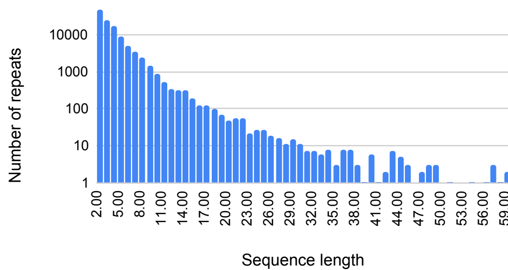


Fig. 8: The histogram of the frequency of repeats (y-axis) vs. sequence length (x-axis). Two-instruction sequences repeat most frequently; the longer the sequence length, the fewer the number of repeats.

histogram). This emphasizes that the machine outlining does not need to go beyond a basic block; the average basic block length is 11.5 instructions in the `UberRider` app, which covers more than 99% of the candidates.

(3) Language and runtime features related to reference counting and memory allocation are the cause of the most frequently repeated patterns The top few frequently appearing patterns in Listings 1-6 are all related to language and the runtime specifics — reference counting and memory allocation of Swift and Objective-C. *Repetitions of this nature cannot be identified at high-level IR abstractions since they are captured within a single IR instruction.*

Since both Swift and Objective-C are reference counted [50], instructions to increment (`swift_retain` and `objc_retain`) and decrement (`swift_release` and `objc_release`) reference are highly frequent. Consider Listing 1 as an example; the first instruction moves the value present in register `$x20` to register `$x0` by performing a logical OR operation (`ORR` instruction) with the zero register `$xzr`. The second instruction (`BL`) invokes `swift_release`, which decrements the reference count of the heap object held in the argument `$x0`. In this example, the pointer to the heap object was originally present in `$x20` (source register), but it had to be moved to `$x0` (destination register) to meet the calling convention [51], [52], which expects the first argument in `$x0`.

Register assignment [4] choices can lead to many repeated patterns — for example, Listings 1 and 2 differ only in their source registers. Over the entire program binary, these patterns can occur many times, which results in repetitions of the pattern. There are many possible targets for a function call instruction, and hence each one contributes to a unique 2-instruction pattern. Finally, the callee can expect more than one argument (e.g., `swift_allocObject` [50] in Listing 3 expects three arguments); hence, the destination register can also be different and be reordered by the *instruction scheduler*, which also contributes to several 2-instruction patterns.

Listing 7 shows a frame setup sequence in `AArch64`; eight callee-saved registers `x19-26` are effectively pushed onto the stack using four `STP` instructions [32], where each instruction stores a pair of registers to contiguous memory pointed to by the stack pointer `$sp`. This sequence repeats about 7K times in the `UberRider` app. Listing 8 shows an analogous frame destruction sequence where the same set of registers are popped from the stack using four `LDP` instructions [32], where each instruction loads a pair of registers from contiguous memory pointed to by `$sp`.

(4) The generous use of novel high-level language features and their corresponding code generation contribute to certain

```
1  $x0 = ORRXrs $xzr, $x20
2  BL swift_release
```

Listing 1: swift_release (46.3K repeats)

```
1  $x0 = ORRXrs $xzr, $x19
2  BL swift_release
```

Listing 2: swift_release (21.8K repeats)

```
1  $w2 = ORRWri $wzr, 2
2  BL swift_allocObject
```

Listing 3: swift_allocObject (64.3K repeats)

```
1  $x0 = ORRXrs $xzr, $x20
2  BL objc_release
```

Listing 4: objc_release (35.2K repeats)

```
1  $x0 = ORRXrs $xzr, $x20
2  BL objc_retain
```

Listing 5: objc_retain (29.5K repeats)

```
1  $x0 = ORRXrs $xzr, $x20
2  BL swift_retain
```

Listing 6: swift_retain (24.4K repeats)

```
1  $sp = STPXpre $x26, $x25,
        $sp(tied-def 0), -10
2  STPXi $x24, $x23, $sp, 2
3  STPXi $x22, $x21, $sp, 4
4  STPXi $x20, $x19, $sp, 6
```

Listing 7: Frame setup (7K repeats)

```
1  $x20, $x19 = LDPXi $sp, 6
2  $x22, $x21 = LDPXi $sp, 4
3  $x24, $x23 = LDPXi $sp, 2
4  $sp, $x26, $x25 = LDPXpost
        $sp(tied-def 0), 10
```

Listing 8: Frame destroy (7K repeats)

```
1 func ul<T: Sequence>(collection: T,
2                    closure: T.Iterator.Element){
3 ▶ evaluate("ul", {for i in collection {closure(i)}})
4 }
5 func table<T: Sequence>(collection: T,
6                        closure: T.Iterator.Element){
7 ▶ evaluate("table", {for i in collection {closure(i)}})
8 }
9 func tbody<T: Sequence>(collection: T,
10                       closure: T.Iterator.Element){
11 ▶ evaluate("tbody", {for i in collection {closure(i)}})
12 }
13 func evaluate(node: String, closure: Closure){
14    ...
15 ▷    closure()
16    ...
17    if let idd = idd { globalMap["id"] = idd }
18    if let dir = dir { globalMap["dir"] = dir }
19    ··· 124 such assignments ···
20 }
```

Listing 9: The closures on lines 3, 7, and 11, result in creating three instances of `evaluate` where the closure is expanded on line 15. The next 124 assignments starting at line 17 form the longest repeating straightline sequence among the three instances.

very long undesirable repeated patterns We elaborate more on this with two examples.

Closure specialization. Listing 9 shows the Swift code skeleton for the *longest repeating pattern* in the `UberRider` app, which is 279 instructions long. The pattern repeats three times. Our analysis attributes it to a code in the third-party HTTP server Swifter [53] used in the app. Below we describe how this code is generated by the Swift compiler from `closure`, which is a high-level language construct.

There are three generic functions `ul`, `table`, and `tbody`, each of which internally invokes the function `evaluate` with two parameters: a `node` string and a `closure`. The function `evaluate` instantiates the closure followed by a series of 124 updates to the `globalMap`. While compiling this code, the Swift compiler hoists all the 124 nullness-checks in lines starting from 17 (i.e., `if let idd = idd ...`) resulting in a very long basic block consisting of instruction to update the `globalMap` 124 times. The compiler also specializes `evaluate` to create three copies based on the three generic functions (lines marked with ▶). This repeat of 124 updates to the `globalMap` forms the longest repeating pattern.

```
1 final public class MyClass: Model, Equatable {
2    public let uuid: MyClassUUID
3    public let dest: Location
4    ... 118 such fields ...
5    public let title: String
6    public init(json: JSON) throws {
7       uuid = try json.getString(at: "uuid")
8       dest = try json.getString(at: "dest")
9          ... 118 such initialization ...
10      title = try json.getString(at: "title")
11   }}
```

Listing 10: A typical idiom in Swift to construct an object by deserializing from JSON. The try expression can throw an error.

$O(N^2)$ code blow-up in out-of-SSA from *try* expressions. Listing 10 shows a common idiom [54], [55] recommended by Swift to use the `try` expressions to deserialize JSON data and assign to properties of a class. In this example, the class `MyClass` contains 118 properties, which are initialized from a JSON object. The initialization happens via `try` expressions, which throw `Error` if the property is not found in the incoming JSON object. In Swift, a throwing function propagates errors to the enclosing scope. Figure 9 depicts the control-flow graph at LLVM-IR level for Listing 10. Each try expression generates two basic blocks: one normal execution block (labeled $T\#$) and another error block (labeled $B\#$). The $T\#$ blocks not only initialize a property but also update its reference counting (not shown). If all try expressions succeed, the code returns successfully after block $T118$. The $B\#$ blocks and their reachable blocks handle the error scenario.

In case of an error, the control branches to the relevant exception handling arm ($B\#$ blocks). The exception handling block sets an appropriate `swift.error` object and then branches to a common basic block marked as L. The block L has 118 incoming edges, one for each try expression. Depending on where the error occurred, reference is decremented for the already initialized properties. To accomplish this goal, the `IRGen` component of the Swift compiler (described in §II-A) creates 118 temporary variables (marked with suffix `Init` in block L). These variables take a true/false value based on the edge taken to reach L. If a variable is set to true, then the corresponding property's reference will be decremented. For example, if an error occurred while parsing JSON in block $T2$, the control transfers to $B2$, and in L, `%uuidInit` will be true having successfully created in $T1$. However, `%dest` and all the subsequent variables will be set to `false`. In general, for a sequence of N `Init` temporaries, `Init1 ... InitN`, if the incoming edge is from block B_i, where $1 \leq i \leq N$, the code generation pattern ensures that all `Init` temporaries in the range $[1..i)$ are set to `true` (because those objects should be released), and all `Init` temporaries in the range $[i..N]$ are set to `false` (because those objects are not created).

Thus, 118 PHI nodes are embedded in L for the properties plus one additional PHI node for the error object. At the end of L, each temporary is checked in the order from the first to the last, and if a temporary's value is set to true, its reference count is decremented, moving on to the next one as shown in the lower part of the Figure 9.

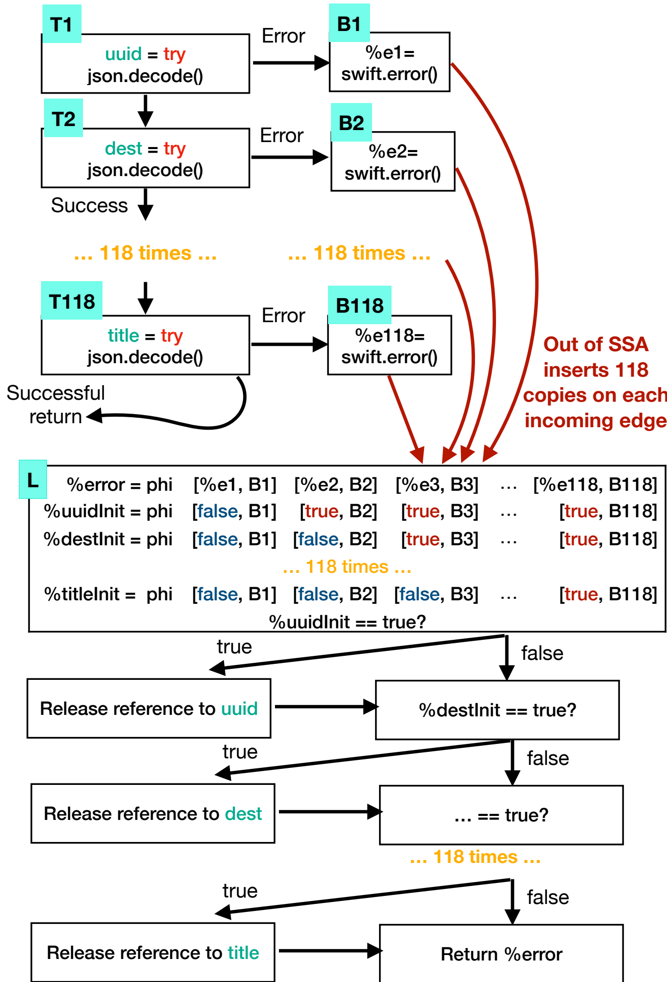


Fig. 9: Generous use of `try` expression in object initialization causes many copy assignments during out-of-SSA translation.

```
1  $w20 = ORRWri $wzr, 0 // copy
2  $w24 = ORRWri $wzr, 0 // copy
3  ... 9 copies ...
4  $w8 = ORRWri $wzr, 0
5  STRXui $x8, $sp, 0 // spill
6  $w8 = ORRWri $wzr, 0
7  STRXui $x8, $sp, 1 // spill
8  ... 25 spills ...
9  $w8 = ORRWri $wzr, 0
10 STRWui $w8, $sp, 24 // spill
```

Listing 11: Out-of-SSA copies and spills due to Swift's `try` clause.

Listing 11 shows a snippet of machine-code instructions that repeat after out-of-ssa translation [56]–[59] of the L block for each PHI. The copy statements are inserted in their predecessor blocks. Some of these copies are performed in registers, and others are spilled. The interesting part is that these copies are inserted in all the predecessor blocks, leading to a major code bloat. The number of copy instructions added after out-of-ssa is $O(N^2)$, where N is the number of try block expressions used for JSON deserializtion. Substrings of these copy instructions are alike among various predecessors, which gives the opportunity for machine-code outlining to save size.

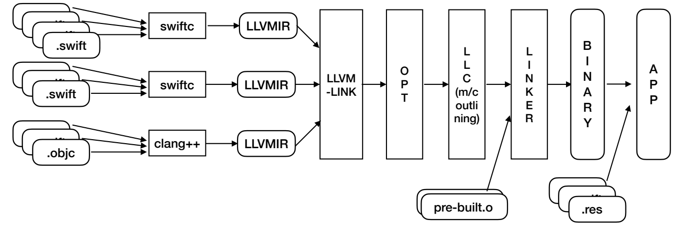


Fig. 10: New build pipeline for exploiting whole-program optimizations in iOS apps.

V. Getting More from Machine Outlining

In this section, we first describe our new whole-program optimization pipeline. Subsequently, we improve the machine outliner to deliver a higher impact.

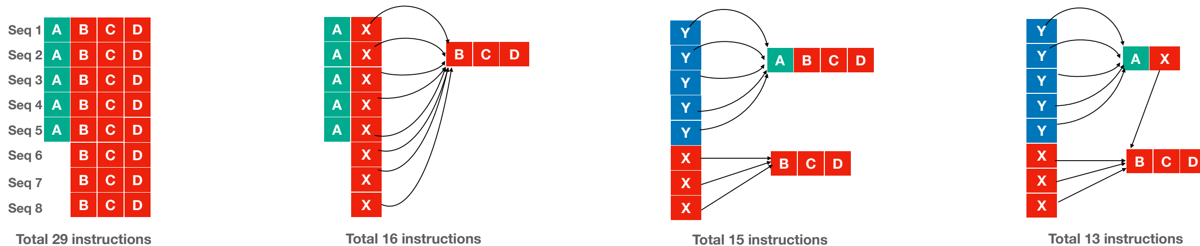
A. New iOS Build Pipeline

Despite machine outlining being enabled in the Swift compiler, the default iOS build pipeline described previously in §II-A is a serious hurdle to making it deliver its full impact. First, since each module is independently compiled into machine code, the total amount of repeated sequences available for outlining within a function is less. Second, the outliner will produce the same outlined function in each module for patterns that repeat in each module — meaning there are several outlined function clones when all machine-code files are linked, defeating the purpose of outlining.

We address this limitation by modifying our build pipeline to adopt whole-program optimization, as shown in Figure 10. The new pipeline produces LLVM IR for each module in lieu of directly producing the machine code. It, then, combines all LLVM-IR files into one large IR file using `llvm-link`. Subsequently, it performs all LLVM-IR level optimizations on this single IR file using `opt`. We then feed the optimized IR to `llc`, which lowers the IR to the target machine code; during this phase, we enable machine outlining on the whole program. This ensures (a) maximum similarity is exploited while identifying candidate machine code sequences, and (b) no outlined function is a clone of another outlined function, which would have been common had we performed only per-module machine outlining. The machine code is finally fed to the system linker along with any pre-compiled machine code to produce the final binary image. §VII-A quantifies the benefits of the whole-program outlining over the default.

B. Improvements to `MachineOutliner`

The greedy machine-outlining algorithm implemented in LLVM squanders a significant size-saving opportunity. We first depict this lost opportunity using an anecdotal example and then show the real sequences from our app. In Figure 11a, two sequences `BCD` and `ABCD` are the two potential patterns to outline. Without the loss of generality, assume no overhead of outlining at the call site or frame overhead for the outlined function. LLVM's `MachineOutliner` chooses `BCD` because it shows the maximum savings in the immediate next step: choosing `BCD` will shrink $8 \times 3 = 24$ instructions into 8



(a) Eight sequences totaling 29 instructions. (b) A greedy choice to outline most profitable pattern reduces the size to 16 instructions. (c) A greedy choice to outline longest sequence reduces the size to 15 instructions. (d) Repeated application of most profitable pattern reduces the size to 13 instructions.

Fig. 11: A demonstration of the suboptimality of greedy outlining and the superiority of repeated outlining.

```
1 $w1 = ORRWri $wzr, 1728
2 $w2 = ORRWri $wzr, 2
3 BL swift_allocObject
```

Listing 12: A 3-instruction pattern has 16K candidates.

```
1
2 $w2 = ORRWri $wzr, 2
3 BL swift_allocObject
```

Listing 13: A 2-instruction substring has 65K candidates.

instructions while introducing a new function of 3 instructions, with a total savings of 13 instructions; in contrast, choosing ABCD will shrink $5 \times 4 = 20$ instructions into 5 instructions and introduce a new function of 4 instructions, with a total savings of only 11 instructions. Outlining BCD, shown in Figure 11b, reduces the code to a total of 16 instructions. Outlining ABCD, however, is more profitable in reality because it not only allows outlining ABCD first but also allows outlining BCD subsequently on the remaining candidates to reduce the total size to 15 instructions, as shown in Figure 11c. However, this cascading effect is not immediately obvious; clearly, the greedy algorithm implemented in LLVM is sub-optimal.

Listing 12 shows a 3-instruction sequence, which repeats 16K times, and Listing 13 shows a 2-instruction sequence, which is a suffix of the 3-instruction pattern but repeats 65K times in the UberRider app. The greedy algorithm prefers the 2-instruction sequence for outlining for immediate profit but fails to exploit size reduction on lengthier sequences.

We address this issue by introducing *repeated* machine outlining in LLVM. The idea of repeated machine outlining is to use the greedy algorithm to choose the next most profitable pattern as before, but, instead of discarding lengthier candidates whose substrings are already outlined, we continue to iteratively apply the same algorithm on the new candidates, which now contain one or more calls to already outlined patterns. Since MachineOutliner relies on up-to-date liveness information, we had to update the candidate's liveness information after the call/branch instructions are introduced, details of which are omitted for brevity. Repeated machine outlining is not the last pass in LLVM; one can further apply other low-level optimizations after applying the repeated outlining. All changes related to repeated machine outlining are upstreamed to LLVM [60], [61].

The repeated outlining offers practical benefits over the default greedy algorithm. Going back to our example, Figure 11d shows that the sequence AX can be outlined during the second repetition of outlining; the final size is 13 instructions — better than both alternatives. The number of repetitions should be tunable. Our evaluation shows that our app converges to an optimal code size after five rounds of machine outlining.

VI. Practical Challenges

In this section, we describe a few challenges in bringing the new whole-program pipeline along with the extended-MachineOutliner to production in the UberRider and other apps. As stated before in §I, compile time, performance, and developer transparency were our topmost considerations.

(1) New pipeline adoption. Overhauling the default build workflow with our custom workflow requires maintaining a local LLVM tool-chain, which required buy-in from several stakeholders, including the Developer Experience , Testing, and Release teams. We tackled this by introducing a configuration flag to either enable or disable the new build pipeline, making it easier to roll-back in the event of outages. Every time a developer checks-in her code in the integration system, a fast debug build is kicked off, which does not use our pipeline; simultaneously, a release build is asynchronously started, which extensively tests our new pipeline using a variety of optimization levels. All our release builds use this new pipeline and add extra 45-minutes to our release builds [within acceptable range]. This strategy of having two separate pipelines ensures developers are unimpacted when performing feature development, bug fixes, or testing in their local environment.

(2) Language interoperability issues. Two LLVM-IR files, one produced from the Swift compiler and another produced from the clang compiler (for Objective-C), could not be merged into a single IR file via llvm-link because of conflicting "Objective-C Garbage Collection" LLVM metadata flag being used by both compilers. Since our app is a mixture of Swift and Objective-C, this support was necessary. Previously the LLVM GCMetadata was a single value that encoded compiler major and minor versions and other bits. Hence, comparing all the bits arising from different compilers led to conflicts. We fixed it by breaking up the LLVM metadata into a set of "attributes"; later the link-phase only inspects the relevant attributes ignoring the compiler that generated it. Thus, we eliminate the conflict. Our fixes are upstreamed to clang and llvm-link [62].

(3) Performance regressions. Despite all our precaution and testing, it was impossible to gauge the performance impact of our changes in production on millions of users using different hardware devices and operating systems and exercising very different use cases in different geographies. Several months after rolling out the outlining optimizations, we noticed an average 10% performance regression. Interestingly, the regression was present whether or not we performed machine outlining but used the new build pipeline. We noticed an increase in page faults related to data. A further investigation pinpointed the problem to the LLVM IR merging in `llvm-link`.

`llvm-link` does not preserve the original order in which the data is present in each constituent module. When numerous modules are merged, the intermixing of data from disparate modules leads to data locality problems. Feature developers typically put all the data needed by a feature in its relevant module and place relevant data together. `llvm-link` destroys this programmer-driven data affinity.

We introduced a new data-layout ordering [63] in `llvm-link` that honors the original module-specific ordering of data present in its constituent IR files even after merging. This optimization eliminated the performance regression.

(4) Debuggability. An outlined function cannot map its instructions back to any specific source location since multiple source locations can map to it. After rolling out the new pipeline, when our developers were investigating bug reports, they were sometimes seeing an `OUTLINED_FUNCTION_ID` on top of their call stacks; they were misunderstanding the failures to be caused by the outlining optimization. None of the failures were in reality related to outlining. Fortunately, the failure reports have the full call stacks rather than just the leaf function. By inspecting a level deeper into the backtrace, the developers were able to debug the failure in their feature code.

VII. Evaluation

In this section, we evaluate the impact of the machine outlining, including the new pipeline on the `UberRider` app over a one-year period using the production data obtained from millions of daily users. We additionally assess performance overhead of machine-code outlining on 26 benchmarks written in Swift. We demonstrate the general applicability of our approach on `UberDriver` app, `UberEats` app, the `clang` compiler, and the Linux kernel.

A. Impact of Machine-Code Outlining

Prior Swift compilers did not have machine outlining. Today, `Swift v5.2` enables `MachineOutliner` at per-module level when compiled with `-OSize`. The `UberRider` app built with the default iOS build pipeline, which uses `Swift v5.2`, produces a **145.7MB binary that has 114.5MB code section, which forms the baseline for our final size gains.**

In order to assess the impact of different rounds of outlining when performed on per-module, we extended the default build pipeline that can both disable outlining and repeat machine outlining per-module. Figure 12 depicts our empirical findings. The x-axis marked as `None` is produced by disabling machine

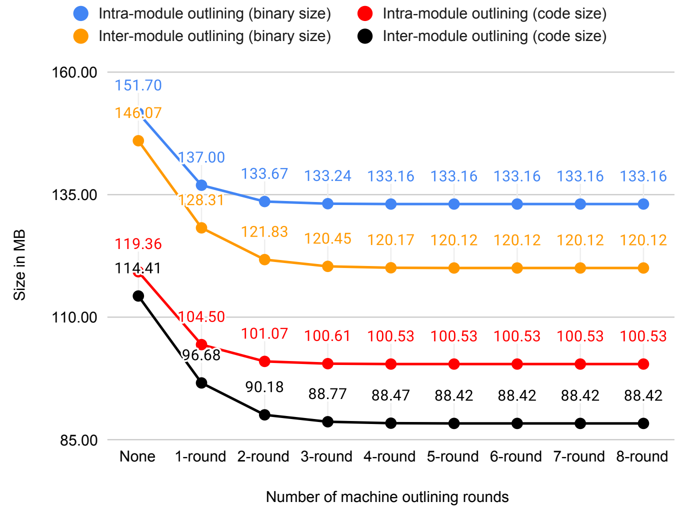


Fig. 12: Comparison of size reduction (a) code section vs. full app binary, (b) different rounds of machine outlining repeats, and (c) whole-program vs. intra-module machine outlining.

outlining, however all other size-reducing optimizations in LLVM are enabled. The rest of the points on the x-axis progressively increase rounds of machine outlining.

First, comparing the whole binary size (the top two lines) with the code size (the bottom two lines) shows that the app binary size reduces proportionally with the code section size because of repeated outlining. Five rounds of machine outlining in our new build pipeline produces a 120.1MB binary, which reduces the *binary size* by 17.6% compared with the default pipeline's 145.7MB. The same produces a *code section* of 88.4MB, which is 22.8% smaller compared with 114.5MB in the default pipeline. Out of the 22.8% code size savings, 27% (7% points) is derived from repeated machine outlining.

Second, there is a continued but diminishing size reduction with an increased number of machine outlining rounds. Also, the gains for the intra-module outlining plateau sooner than the inter-module outlining. Three rounds of outlining extract most of the size benefits. Beyond five rounds, there is no benefit at all, but the initial few rounds cannot be discounted. We chose five rounds as the default for the `UberRider` app.

Third, comparing the pair of bottom two lines, it is clear that inter-module (whole program) repeated machine outlining significantly outperforms intra-module outlining. At five rounds of repeats, the whole-program machine outlining delivers 88.42MB code size, whereas doing the same on only individual modules delivers a 100.53MB (13.7%) code size increase.

Table II shows compile-time statistics on the number of machine code sequences outlined (row 1), the number of new functions created (row 2), and the total size of code needed for the outlined functions (row 3) at each repetition of machine outlining. Five rounds of outlining eliminates 4.7 million candidates at the cost of creating 259K more functions, which consume 3.53MB of code.

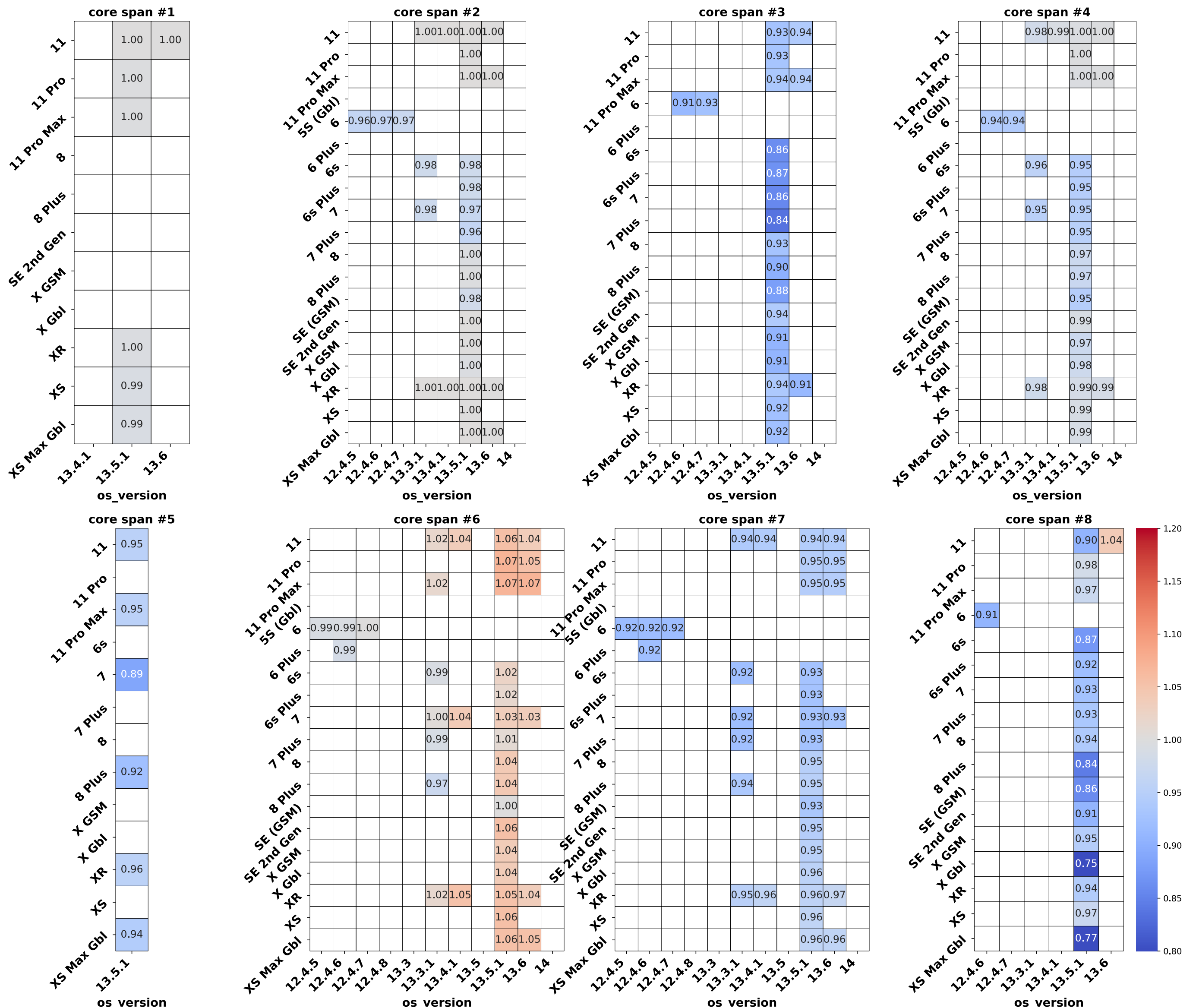


Fig. 13: Performance comparison of five rounds of machine-code outlining in the new build pipeline against the default iOS build pipeline. A cell with a red tinge implies performance regression (unfavorable); a cell with with a blue tinge shows performance improvements (favorable).

TABLE II: OUTLINING STATISTICS AT DIFFERENT LEVELS OF REPEATS.

Metric	rounds of outlining				
	1	2	3	4	5
# sequences outlined ($\times 10^6$)	3.08	4.30	4.62	4.70	4.71
# functions created ($\times 10^5$)	1.15	2.03	2.44	2.57	2.59
Bytes consumed by outlined functions ($\times 10^6$)	1.69	2.80	3.33	3.50	3.53

B. *Production Performance Data*

Outlining may degrade performance due to extra branch/call overhead. However, performance gains are also possible because of the reduced instruction footprint. The `UberRider` app is intensive on the User Interface (UI), and our code footprint is heavy. A large fraction of the code is run only once in a typical usage scenario — there is no single "hotspot" code, unlike HPC-style code.

Figure 13 shows the heatmaps for several critical use cases (named core-spans) identified by the `UberRider` app development team. The rows in each span represent different hardware versions, and the columns represent different OS versions. Since the data from production can be noisy, we populate only those cells with $> 25K$ samples both before and after optimization. The value in each cell is the ratio of the 50th percentile (P50) time to execute the span with our whole-program 5-rounds of repeated machine-code outlining, divided by the time to execute the same span without the optimization; hence, a value greater than 1.0 implies performance regression, and a value less than 1.0 shows performance improvement. The data-layout optimization described in Section VI is already

TABLE III: AVERAGE EXECUTION TIME OF CORE SPANS.

SPAN	Baseline (sec)	Optimized (sec)
SPAN1	4.7s	3.9s
SPAN2	2.1s	2.0s
SPAN3	6s	5.6s
SPAN4	2.1s	2.1s
SPAN5	1.3s	1.3s
SPAN6	.64s	.66s
SPAN7	1.1s	1.1s
SPAN8	2.6s	1.8s

enabled in our optimized versions. Table III shows the execution time of these core spans averaged over all OSes and devices.

P50 focuses on the central tendency. The other percentile values (e.g., p75, p95) from production showed similar trends. About 3% of dynamic instructions execute outlined instructions in the `UberRider` app.

A handful of spans show some performance improvements. On average there is 3.4% performance gain, and in the best case it is 25% for span 8, on `13.5.1` OS on `iPhone X Gbl` device. There are multiple factors in play; outlining leads to a smaller instruction footprint and hence possibly less icache and iTLB pressure, but it introduces slightly more instructions to accomplish the same quantity of work. We observed a 4% increase in instructions per cycle (IPC) with machine outlining compared to no outlining, which is commensurate with the 3.4% performance gain. Span 6 shows some regression. It is the shortest span with only 0.64 seconds of execution. Notice, however, that the slowdown is negligible. With the diversity in OSes and hardware, continually changing software, and other region-specific experimental flags, it was not feasible to isolate improvements or regressions to a particular hardware subsystem in an out-of-order, multi-issue, deeply pipelined processor, where the penalty matters if it leads to stall cycles.

In Figure 13, we notice more blue cells indicating overall performance gains. Overall, we see a geometric mean performance gain of 3.4% due to our new pipeline and optimization. Given the volume of real-world data used in the evaluation, we are confident about the conclusions derived and convinced that machine outlining, when performed with a whole-program pipeline, not only saves app binary size by 23% but also mildly improves performance by 3.4% for iOS mobile applications with a large code footprint and few code hotspots.

C. Build Time

We evaluate the compile time on a 10-core iMac Pro (2017) equipped with a 64GB DDR4 running MacOS `10.15.6`. The default pipeline builds the app in 21 minutes; the new pipeline with no machine outlining takes 53 minutes, which includes about 7 minutes of `llvm-link`, 14 minutes `opt`, 11 minutes of `llc` and 3 minutes of the system linker. One round of outlining takes about 7 minutes in `llc`, and two rounds take 9 minutes. Each additional round adds progressively less extra time, usually under 30 seconds. Overall, five rounds of outlining builds in 66 minutes — a 45-minutes addition to the baseline.

D. Lifelong Code-Size Savings

Our new pipeline finds more opportunities for binary-size reduction in a continuous development environment. We refer the reader to Figure 1 in §I at the beginning of this paper to observe the impact of repeated machine-code outlining on our app code bytes. In this figure, the baseline (blue) code size is already optimized for size, but it uses per-module optimization and does not have repeated machine outlining (which represents the default iOS pipeline).

The code-size growth for the baseline fitted with the linear regression line has a slope of 2.7 (96% confidence). The code-size growth with our optimizations (red line) has a slope of 1.37 (98% confidence). Hence, we reduce the code size growth by about $2\times$. We believe this "life-long" code-size impact is a significant benefit of the optimizations we developed.

E. Generality of Repeated Machine-Code Outlining

1) Other iOS apps: Following the success of the `UberRider` app, we rolled out whole-program machine-code outlining with five rounds of repeats to `UberDriver` and `UberEats` apps. Both apps are used daily by millions of users worldwide. The `UberDriver` app is 2.2 million lines of code with 77% swift and 23% objective-C; the `UberEats` app is 2.1 million lines of code with 66% swift and 34% objective-C. The insights presented in Section IV for the `UberRider` app — short sequence of repeats, power law of repetition frequency, language features causing repeats, LLVM out-of-SSA causing copies, and importance of repeated machine outlining — translate to these apps also. `UberDriver` and `UberEats` apps shrank their code size by 17% and 19%, respectively, using our optimizations.

2) Non-iOS programs: We also applied our techniques to the `Clang v9.0.0` compiler and the Android `v4.19` Linux kernel. We observed short sequences that move multiple registers to/from memory to be common; the frequency distribution showed the power-law; the register movement to setup calling convention often appeared as top outlining candidates; in the Linux kernel, the function epilogue to check stack smashing attack is a common repeating code pattern. Five-rounds of machine outlining reduced the code size of Linux kernel and `Clang` by 14% and 25%, respectively.

3) Benchmarks: We collected a set 26 swift benchmarks that implement popular algorithms [64]. The benchmarks are small and single-module; hence, they do not represent a typical use case scenario where we recommend applying our optimizations. Evaluating these benchmarks aims to understand the performance overhead when the outlining happens in frequently executed code paths. Table IV shows the performance degradation after five rounds of outlining on these benchmarks on the previously mentioned iMac Pro system. The average slowdown is only 1.63%, and there are several instances of speedups. The Dijkstra's algorithm incurs a 10.81% slowdown.

We also created a pathological case where the program consisted of a long-running loop with 2-instruction body, where the body was replaced via outlining; it showed only an

TABLE IV: PERFORMANCE OVERHEAD OF FIVE ROUNDS OF MACHINE-CODE OUTLINING ON SWIFT BENCHMARKS. A NEGATIVE OVERHEAD MEANS A SPEEDUP.

Benchmark	%overhead	Benchmark	%overhead
BFS	2.76	JSON	1.13
Boyer-Moore-Horspool	2.90	Knuth-Morris-Pratt	-1.49
BucketSort	-0.12	LCS	-3.40
ClosestPair	0.48	LRUCache	0.69
Combinatorics	1.05	OctTree	0.02
CountingSort	-3.42	QuickSort	1.54
CountOccurences	-0.64	RedBlackTree	6.54
DFS	1.75	RunLengthEncoding	0.42
Dijkstra	10.81	SimulatedAnneaning	1.39
EncodeAndDecodeTree	1.69	SplayTree	7.06
GCD	3.19	StrassenMM	1.99
HasTable	0.70	TopologicalSort	2.19
Huffman	3.12	Z-algorithm	0.06
		Average	1.63

8.67% slowdown. Outlined branches are predictable by modern hardware, and the cost is largely hidden in the pipeline.

VIII. CONCLUSIONS AND FUTURE WORK

We performed a systematic analysis of the UberRider app, a production iOS application and showed numerous repeating patterns of machine code and pinpointed their causes to high-level language features. Machine outlining, when applied at the whole-program level, reduces app binary size significantly. We enhanced LLVM's machine outlining with *repeated outlining* to reduce the code size further. Our optimizations are in production, used by millions of daily-users, and have been instrumental in keeping the app size under control. The benefits of our optimizations grow over time, making them highly effective for code-size reduction and desirable in a fast-growing code base. We evaluated the impact of our optimizations on the app performance and compile time; we showed no performance regressions in large-scale deployments and alleviated any compile-time impact for hundreds of developers.

Code-size optimization has been at the heart of compiler technology for several decades, but less work has been done to detect *missed opportunities* in code size. Observing replicated machine-code sequences at the whole-program level opens a new avenue to pinpoint and quantify repeated code patterns and attribute them to various layers of code transformation. We have also found the applicability of our findings on UberDriver and UberEats iOS apps and also on the open-source desktop and server programs such as the clang compiler and the Linux kernel. Our future work involves exploring (1) semantic equivalence of machine-code sequences, (2) interactions between instruction scheduling, register assignment, and machine-code outlining, and (3) layout optimization on the outlined code.

APPENDIX A
ARTIFACT EVALUATION

A. Abstract

We make the LLVM compiler changes available with this artifact. Since the UberRider app is proprietary, its source code or binary cannot be released. In lieu of the UberRider app, we provide a set of Swift algorithmic benchmark programs and precompiled Android v4.19 Linux kernel and Clang 9.0.0 compiler LLVM bit codes for exercising our algorithm.

We show the impact of the compiler changes on a set of Swift benchmarks and Android v4.19 Linux kernel and Clang 9.0.0 compiler applications. There are three dimensions to the artifact.

1) Apply our changes to LLVM and produce a functional LLVM toolchain.
2) Compare the performance of a suite of Swift benchmarks with and without the machine outlining optimizations; in this case, the code size is not important because the benchmarks are very small. Please note, as described in the paper, the real high impact of machine outlining and repeated machine outlining is when we have multiple compilation units analogous to many swift modules (see next bullet item).
3) Apply our code-size optimizations and demonstrate size savings (with and without outlining) on large open-source code; we pick the clang compiler and android Linux code (already compiled into LLVM bitcode). Here we show how repeated machine code outlining yields significant size reductions. This is comparable to what we see on the UberRider app.

These steps are automated and the scripts can be easily extended to exercise more benchmarks and apps. The only externally visible commandline flag is -outine-repeat-count=<uint> to the LLVM's llc tool, which controls how many times to perform outlining.

B. Artifact Checklist

- **Algorithm:** Repeated machine-code outlining.
- **Program:** LLVM.
- **Run-time environment:** MacOS and Xcode 11.6.0.11E70.
- **Hardware:** Apple MacBook / MacMini / iMac series.
- **Metrics:** Running-time comparison and size comparison.
- **Output:** Speedup/slowdown and size savings on the console.
- **Experiments:** Size and speed comparisons.
- **How much disk space required (approximately)?:** 6 GB.
- **How much time is needed to prepare workflow (approximately)?:** 5 minutes.
- **How much time is needed to complete experiments (approximately)?:** 3 hours.
- **Publicly available?:** Yes.
- **Code licenses (if publicly available)?:** University of Illinois/NCSA.
- **Data licenses (if publicly available)?:** 2016 Matthijs Hollemans and contributors and GPL.
- **Workflow framework used?:** No.
- **Archived (provide DOI)?:** 10.5281/zenodo.4281687

C. Description

1) How Delivered: Uploaded to https://doi.org/10.5281/zenodo.4281687.

2) Hardware Dependencies: A reasonably new Apple Mac laptop running on an x86-64 machine; any other Mac system such as MacMini or iMac will also do. Exercising the UberRider app would have required an AArch64 iOS device and several app signing tokens, which is not possible outside of the industrial setting.

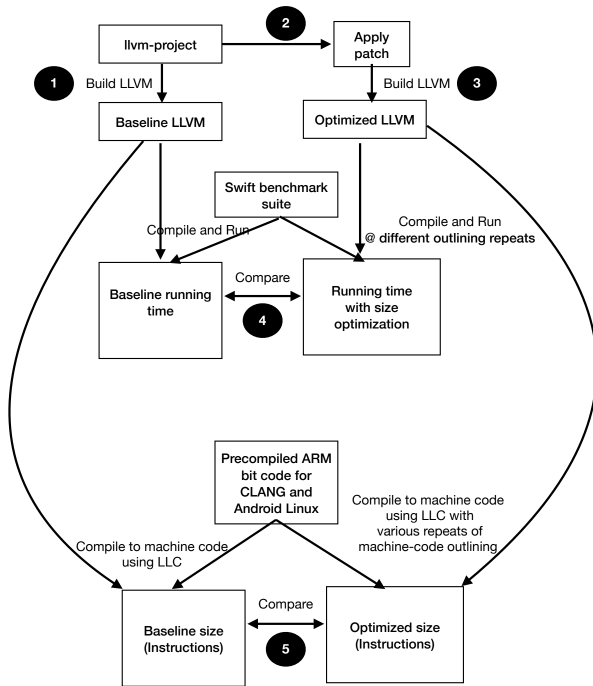


Fig. 14: Workflow of evaluating the artifact. Numbers in the black circle is the order to follow evaluation. 1) build baseline LLVM, 2) apply our repeated machine-code outlining patch and build an optimized version of LLVM 3) run Swift benchmarks suites compiled with both versions of LLVM and compare their running time, 4) compile `clang 9.0.0` and Android `4.19` Linux from bit code to `AArch64` machine code using both compilers (applying a different number of outlining repeats when using the optimized version) and compare the sizes.

3) Software Dependencies:

- `Xcode 11.6.0.11E70`. Any other Xcode 11.0 series should be fine, but do not use `Xcode 12+` since our distributed version of LLVM will become older for those versions. Also, look up https://xcodereleases.com/ to make sure your OS is up-to-date with the Xcode version you choose.
- `Ninja`
- `cmake`

4) Data Sets:

- Randomly picked benchmarks from https://github.com/raywenderlich/swift-algorithm-club and slightly modified to run long enough to measure performance.
- `clang-9` LLVM bitcode file compiled for size. It is compiled for an `AArch64` Linux machine.
- `Android 4.19 Linux` LLVM bitcode file compiled for size. It is compiled for an `AArch64` architecture.

D. Installation

```
1. Install Xcode 11.6.0.11E70.
2. brew install ninja
3. brew install cmake
4. wget -o artifact.tgz \
 https://zenodo.org/record/4281687/files/
 artifact.tgz?download=1
5. tar zxf artifact.tgz
6. cd artifact
```

E. Experiment Workflow

Figure 14 depicts our experimental workflow.

F. Artifact Organization

1) `llvm-project.tgz` is the baseline LLVM project obtained from https://github.com/llvm/llvm-project/ at the commit version `3bd4b5a925bd5bd5a5498d8d84596ec099e9c198`.
2) `patchfile.patch` is our repeated machine outlining patch.
3) `install.sh` unarchives `llvm-project.tgz`, `benchmarks.tgz`, and `apps.tgz`.
4) `benchmarks.tgz` once uncompressed into `benchmarks` directory, contains Swift benchmarks.
5) `apps.tgz` once uncompressed into `apps` directory, contains `apps/android_linux.arm64.bc` and `apps/clang9.arm64.bc`, which are respectively bitcode files for Android 4.19 Linux and `clang 9.0.0`, respectively.
6) `runme.sh` is the high-level automation script.
7) `*/clean.sh` and `*/run.sh` scripts clean and run respective directories where they are present.

G. Evaluation and Expected Result

Running the automation script should run everything in one command.

```
./runme.sh
```

This script will follow the workflow in Figure 14.

1) It unarchives the `.tgz` files into respective directories using `install.sh`
2) It builds the baseline LLVM into `llvm-project/BASELINE`.
3) It applies the patch `patchfile.patch`.
4) It builds the LLVM with our patch into `llvm-project/OPTIMIZED`.
5) It enters the `benchmarks` directory and runs `run.sh` script.
   a) `run.sh` compiles each single `.swift` file benchmark into an LLVM bitcode file with `-Osize` flag.
   b) It performs the standard bitcode-to-bitcode size optimizations using the LLVM `opt` tool.
   c) It performs bitcode-to-machine code generation using the LLVM `llc` tool. This is a crucial step that exercises our changes. This step is repeated using both the `llvm-project/BASELINE` and `llvm-project/OPTIMIZED` LLC versions. We repeat the machine-code outlining 1-5 times producing one machine-code file in each setting.
   d) Links each machine code file into a mach-o executable using `ld`.
   e) Runs the executables ten times each (for both baseline and optimized) and shows performance slowdown of outlined versions with respect to the baseline (negative number means speedup). It also shows the size savings with respect to the baseline; however, the size savings for benchmarks are inconsequential. Notice that the runtime slowdown is typically very small, which shows that the repeated outlining did not significantly degrade the performance even when outlined code was highly executed in these synthetic benchmarks. You may compare the results output to the screen with the example output present at `benchmarks/expected_results.txt`.
6) It enters the `apps` directory and runs `run.sh` script.
   a) `run.sh` compiles each of the `.bc` files into machine code using the two variants of `llc`. We repeat the machine-code outlining 1-5 times producing one machine-code file in each setting.
   b) For each app, it outputs the number of machine code instructions generated, the number of candidate machine-code sequences outlined, the number of new outlined

functions created, and finally the percentage size saving with respect to no machine-code outlining. Notice how the size savings increase with each repeated machine outlining, which showcases the strength of repetition of machine-code outlining. Note, that the saving is computed based on the number of instructions, which is fixed-width in `AArch64`; for a varying-width instruction architectures, one needs to use other techniques such as looking into the `.text` section of the binary. You may compare the results output to the screen with the example output present at `apps/expected_results.txt`

H. Experiment Customization

One can add more Swift benchmarks to `benchmarks/run.sh` file and obtain more results. One can add more LLVM bitcode files to `apps/run.sh` file and obtain more results. Note that if the bitcode is not for `AArch64` architecture, adjustments to the flag passed to `llc` may be necessary. One can add more number of outlining repeats by changing the value passed to `-outine-repeat-count` flag in both `apps/run.sh` and `benchmarks/run.sh`.

References

[1] I. Mansoor, "App Revenue Statistics (2019)," https://www.businessofapps.com/data/app-revenues/t, Jul 2020.

[2] "App Store," https://www.apple.com/ios/app-store/, Apple.

[3] A. Beszédes, R. Ferenc, T. Gyimóthy, A. Dolenc, and K. Karsisto, "Survey of code-size reduction methods," *ACM Comput. Surv.*, vol. 35, no. 3, p. 223–267, Sep. 2003. [Online]. Available: https://doi.org/10.1145/937503.937504

[4] L. Torczon and K. Cooper, *Engineering A Compiler*, 2nd ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007.

[5] J. Cocke, "Global Common Subexpression Elimination," *SIGPLAN Not.*, vol. 5, no. 7, p. 20–24, Jul. 1970. [Online]. Available: https://doi.org/10.1145/390013.808480

[6] R. Kennedy, S. Chan, S.-M. Liu, R. Lo, P. Tu, and F. Chow, "Partial Redundancy Elimination in SSA Form," *ACM Trans. Program. Lang. Syst.*, vol. 21, no. 3, p. 627–676, May 1999. [Online]. Available: https://doi.org/10.1145/319301.319348

[7] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, "Compilers: Principles, techniques, and tools second edition," 2007.

[8] Thomas J. Watson IBM Research Center. Research Division and Allen, FE and Cocke, J, "A catalogue of optimizing transformations," *IBM Technical Reports*, 1971.

[9] B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Global value numbers and redundant computations," in *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '88. New York, NY, USA: Association for Computing Machinery, 1988, p. 12–27. [Online]. Available: https://doi.org/10.1145/73560.73562

[10] M. N. Wegman and F. K. Zadeck, "Constant propagation with conditional branches," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 13, no. 2, pp. 181–210, 1991.

[11] M. M. Chabbi, J. M. Mellor-Crummey, and K. D. Cooper, "Efficiently exploring compiler optimization sequences with pairwise pruning," in *Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era*, ser. EXADAPT '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 34–45. [Online]. Available: https://doi.org/10.1145/2000417.2000421

[12] L. Almagor, K. D. Cooper, A. Grosul, T. J. Harvey, S. W. Reeves, D. Subramanian, L. Torczon, and T. Waterman, "Finding effective compilation sequences," in *Proceedings of the 2004 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, ser. LCTES '04. New York, NY, USA: Association for Computing Machinery, 2004, p. 231–239. [Online]. Available: https://doi.org/10.1145/997163.997196

[13] K. D. Cooper, P. J. Schielke, and D. Subramanian, "Optimizing for reduced code space using genetic algorithms," in *Proceedings of the ACM SIGPLAN 1999 Workshop on Languages, Compilers, and Tools for Embedded Systems*, ser. LCTES '99. New York, NY, USA: Association for Computing Machinery, 1999, p. 1–9. [Online]. Available: https://doi.org/10.1145/314403.314414

[14] P. A. Kulkarni, D. B. Whalley, G. S. Tyson, and J. W. Davidson, "Exhaustive optimization phase order space exploration," in *Proceedings of the International Symposium on Code Generation and Optimization*, ser. CGO '06. USA: IEEE Computer Society, 2006, p. 306–318. [Online]. Available: https://doi.org/10.1109/CGO.2006.15

[15] J. Ernst, W. Evans, C. W. Fraser, T. A. Proebsting, and S. Lucco, "Code compression," in *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*, ser. PLDI '97. New York, NY, USA: Association for Computing Machinery, 1997, p. 358–365. [Online]. Available: https://doi.org/10.1145/258915.258947

[16] T. J. Edler von Koch, B. Franke, P. Bhandarkar, and A. Dasgupta, "Exploiting function similarity for code size reduction," in *Proceedings of the 2014 SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems*, ser. LCTES '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 85–94. [Online]. Available: https://doi.org/10.1145/2597809.2597811

[17] W.-K. Chen, B. Li, and R. Gupta, "Code compaction of matching single-entry multiple-exit regions," in *Proceedings of the 10th International Conference on Static Analysis*, ser. SAS'03. Berlin, Heidelberg: Springer-Verlag, 2003, p. 401–417.

[18] J. Lau, S. Schoenmackers, T. Sherwood, and B. Calder, "Reducing code size with echo instructions," in *Proceedings of the 2003 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, ser. CASES '03. New York, NY, USA: Association for Computing Machinery, 2003, p. 84–94. [Online]. Available: https://doi.org/10.1145/951710.951724

[19] H. Massalin, "Superoptimizer: A look at the smallest program," in *Proceedings of the Second International Conference on Architectual Support for Programming Languages and Operating Systems*, ser. ASPLOS II. Washington, DC, USA: IEEE Computer Society Press, 1987, p. 122–126. [Online]. Available: https://doi.org/10.1145/36206.36194

[20] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, ser. CGO '04. USA: IEEE Computer Society, 2004, p. 75.

[21] J. Caldwell and S. Chiba, "Reducing calling convention overhead in object-oriented programming on embedded arm thumb-2 platforms," in *Proceedings of the 16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, ser. GPCE 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 146–156. [Online]. Available: https://doi.org/10.1145/3136040.3136057

[22] S. K. Debray, W. Evans, R. Muth, and B. De Sutter, "Compiler techniques for code compaction," *ACM Trans. Program. Lang. Syst.*, vol. 22, no. 2, p. 378–415, Mar. 2000. [Online]. Available: https://doi.org/10.1145/349214.349233

[23] T. Glek and J. Hubicka, "Optimizing real world applications with GCC Link Time Optimization," 2010.

[24] B. Schwarz, S. Debray, G. Andrews, and M. Legendre, "Plto: A link-time optimizer for the Intel IA-32 architecture," in *Proc. 2001 Workshop on Binary Translation (WBT-2001)*, 2001.

[25] B. De Sutter, L. Van Put, D. Chanet, B. De Bus, and K. De Bosschere, "Link-Time Compaction and Optimization of ARM Executables," *ACM Trans. Embed. Comput. Syst.*, vol. 6, no. 1, p. 5–es, Feb. 2007. [Online]. Available: https://doi.org/10.1145/1210268.1210273

[26] D. Chanet, B. De Sutter, B. De Bus, L. Van Put, and K. De Bosschere, "System-wide compaction and specialization of the linux kernel," in *Proceedings of the 2005 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, ser. LCTES '05. New York, NY, USA: Association for Computing Machinery, 2005, p. 95–104. [Online]. Available: https://doi.org/10.1145/1065910.1065925

[27] H. He, J. Trimble, S. Perianayagam, S. Debray, and G. Andrews, "Code compaction of an operating system kernel," in *International Symposium on Code Generation and Optimization (CGO'07)*, 2007, pp. 283–298.

[28] N. Pitre, "Shrinking the kernel with link-time optimization," https://lwn.net/Articles/744507/, January 2018.

[29] P. Zhao and J. N. Amaral, "Function outlining and partial inlining," in *17th Symposium on Computer Architecture and High Performance Computing (SBAC-PAD 2005), 24-27 October 2005, Rio de Janeiro, Brazil*. IEEE Computer Society, 2005, pp. 101–108. [Online]. Available: https://doi.org/10.1109/CAHPC.2005.26

[30] Z. Peng and A. Jose Nelson, "Function outlining," https://pdfs.semanticscholar.org/6c72/2c58232816b74e23ebd60f9782073c29699b.pdf.

[31] TIOBE Company, "TIOBE index for august 2020," https://www.tiobe.com/tiobe-index/.
[32] "ARM A64 Instruction Set Architecture," https://static.docs.arm.com/ddi0596/a/DDI_0596_ARM_a64_instruction_set_architecture.pdf, ARM Ltd.
[33] "RxSwift: ReactiveX for Swift," https://github.com/ReactiveX/RxSwift, RxSwift Team.
[34] "SnapKit: A Swift Autolayout DSL for iOS and OS X," https://github.com/SnapKit, SnapKit Team.
[35] "Swift NIO," https://github.com/apple/swift-nio, Swift-NIO Team.
[36] "Freddy," https://github.com/bignerdranch/Freddy, Freddy Team.
[37] "Swift Protobuf," https://github.com/apple/swift-protobuf, Swift Protobuf Team.
[38] M. K. Ramanathan, L. Clapp, R. Barik, and M. Sridharan, "Piranha: Reducing feature flag debt at uber," in *2020 IEEE/ACM 42nd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 2020, pp. 221–230.
[39] J. Paquette, "Reducing code size using outlining," http://www.llvm.org/devmtg/2016-11/Slides/Paquette-Outliner.pdf.
[40] P. developers, "PMD: An extensible cross-language static code analyzer." https://pmd.github.io/, Oct 2018.
[41] "Swift Programming Language," https://github.com/apple/swift/blob/master/lib/IRGen/Outlining.cpp, Swift Team.
[42] "MergeFunctions pass, how it works," https://llvm.org/docs/MergeFunctions.html, LLVM Compiler Infrasturcture.
[43] R. C. O. Rocha, P. Petoumenos, Z. Wang, M. Cole, and H. Leather, "Function merging by sequence alignment," in *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO 2019. IEEE Press, 2019, p. 149–163.
[44] R. Rocha, P. Petoumenos, Z. Wang, M. Cole, and H. Leather, "Effective Function Merging in the SSA Form," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 854–868. [Online]. Available: https://doi.org/10.1145/3385412.3386030
[45] R. Riddle, "IR Outliner Pass," https://reviews.llvm.org/D53942, Oct 2018.
[46] C. V. Lopes, P. Maj, P. Martins, V. Saini, D. Yang, J. Zitny, H. Sajnani, and J. Vitek, "Déjàvu: A map of code duplicates on github," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, Oct. 2017. [Online]. Available: https://doi.org/10.1145/3133908
[47] R. Rocha, "CGO19FMSA.tar.gz," 12 2018. [Online]. Available: https://figshare.com/articles/software/CGO19FMSA_tar_gz/7473260
[48] "MergeSimilarFunctions," https://reviews.llvm.org/D76522, LLVM Github Monorepo.
[49] "Fractal," https://en.wikipedia.org/wiki/Fractal, Wikipedia.
[50] "The Swift Runtime," https://github.com/apple/swift/blob/master/docs/Runtime.md, Swift Team.
[51] "The Swift Calling Convention," https://github.com/apple/swift/blob/master/docs/ABI/CallingConvention.rst, Swift Team.
[52] "64-Bit Architecture Register Usage," https://github.com/apple/swift/blob/master/docs/ABI/RegisterUsage.md, Swift Team.
[53] "Tiny http server engine written in Swift programming language." https://github.com/httpswift/swifter, Swifter developers.
[54] "Error Handling," https://docs.swift.org/swift-book/LanguageGuide/ErrorHandling.html, Swift Team.
[55] F. Phill, "Encoding and Decoding JSON with Swift 4," https://medium.com/@phillfarrugia/encoding-and-decoding-json-with-swift-4-3832bf21c9a8.
[56] B. Boissinot, A. Darte, F. Rastello, B. D. de Dinechin, and C. Guillon, "Revisiting Out-of-SSA Translation for Correctness, Code Quality and Efficiency," in *Proceedings of the 7th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '09. USA: IEEE Computer Society, 2009, p. 114–125. [Online]. Available: https://doi.org/10.1109/CGO.2009.19
[57] V. C. Sreedhar, R. D.-C. Ju, D. M. Gillies, and V. Santhanam, "Translating out of static single assignment form," in *Proceedings of the 6th International Symposium on Static Analysis*, ser. SAS '99. Berlin, Heidelberg: Springer-Verlag, 1999, p. 194–210.
[58] P. Briggs, K. D. Cooper, T. J. Harvey, and L. T. Simpson, "Practical improvements to the construction and destruction of static single assignment form," *Softw. Pract. Exper.*, vol. 28, no. 8, p. 859–881, Jul. 1998.
[59] F. Rastello, F. d. Ferrière, and C. Guillon, "Optimizing translation out of ssa using renaming constraints," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, ser. CGO '04. USA: IEEE Computer Society, 2004, p. 265.
[60] "Fix incorrect logic in maintaining the side-effect of compiler generated outliner functions," https://reviews.llvm.org/D71217, LLVM Github Monorepo, December 2019.
[61] "Support repeated machine outlining," https://reviews.llvm.org/D71027, LLVM Github Monorepo, December 2019.
[62] "Fix conflict value for metadata Objective-C Garbage Collection in the mix of swift and Objective-C bitcode," https://reviews.llvm.org/D71219, LLVM Github Monorepo, December 2019.
[63] "Preserve the lexical order of global variables during llvm-link merge," https://reviews.llvm.org/D94202, LLVM Github Monorepo, Jan 2021.
[64] W. Ray, "Swift Algorithm Club," https://github.com/raywenderlich/swift-algorithm-clu, 2021, accessed: 2021-01-01.