Optimization of Swift Protocols

RAJKISHORE BARIK, Uber Technologies Inc., USA
MANU SRIDHARAN, University of California, Riverside, USA
MURALI KRISHNA RAMANATHAN, Uber Technologies Inc., USA
MILIND CHABBI, Uber Technologies Inc., USA

Swift, an increasingly-popular programming language, advocates the use of *protocols*, which define a set of required methods and properties for conforming types. Protocols are commonly used in Swift programs for abstracting away implementation details; e.g., in a large industrial app from Uber, they are heavily used to enable mock objects for unit testing. Unfortunately, heavy use of protocols can result in significant performance overhead. Beyond the dynamic dispatch often associated with such a feature, Swift allows for both value and reference types to conform to a protocol, leading to significant boxing and unboxing overheads.

In this paper, we describe three new optimizations and transformations to reduce the overhead of Swift protocols. Within a procedure, we define `LocalVar`, a dataflow analysis and transformation to remove both dynamic dispatch and boxing overheads. We also describe `Param`, which optimizes the case of protocol-typed method parameters using specialization. Finally, we describe `SoleType`, a transformation that injects casts when a global analysis (like type-hierarchy analysis) discovers some protocol variable must have some concrete type. We also describe how these optimizations work fruitfully together and with existing Swift optimizations to deliver further speedups.

We perform elaborate experimentation and demonstrate that our optimizations deliver an average 1.56× speedup on a suite of Swift benchmarks that use protocols. Further, we applied the optimizations to a production iOS Swift application from Uber used by millions of customers daily. For a set of performance spans defined by the developers of the application, the optimized version showed speedups ranging from 6.9% to 55.49%. A version of our optimizations has been accepted as part of the official Swift compiler distribution.

CCS Concepts: • **Software and its engineering → Compilers**.

Additional Key Words and Phrases: swift, protocol, existential containers, virtual method call, boxing/unboxing

ACM Reference Format:
Rajkishore Barik, Manu Sridharan, Murali Krishna Ramanathan, and Milind Chabbi. 2019. Optimization of Swift Protocols. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 164 (October 2019), 27 pages. https://doi.org/10.1145/3360590

1 INTRODUCTION

Swift is a relatively new compiled programming language that was initially developed for programming the iOS and Mac OS X platforms. Since its release, Swift has been gaining in popularity, and

Authors' addresses: Rajkishore Barik, Uber Technologies Inc. San Francisco, CA, USA, rajbarik@uber.com; Manu Sridharan, Computer Science and Engineering, University of California, Riverside, Riverside, CA, USA, manu@cs.ucr.edu; Murali Krishna Ramanathan, Uber Technologies Inc. San Francisco, CA, USA, murali@uber.com; Milind Chabbi, Uber Technologies Inc. San Francisco, CA, USA, milind@uber.com.

2475-1421/2019/10-ART164
https://doi.org/10.1145/3360590

it now has ports for multiple platforms and a version with built-in support for the TensorFlow machine learning framework.[1] Positive features include strong static typing, error handling constructs to prevent crashes after deployment, and automatic memory management via reference counting [SwiftLangDoc 2019]. One of the popular paradigms associated with Swift is protocol-oriented programming [Hoffman 2019], where the design favors composition over inheritance [Gamma et al. 1995]. Unlike inheritance-heavy systems where classes at the top of the hierarchy become monolithic by including unrelated functionalities, the composable nature of protocols enables these large monolithic classes to be broken into smaller components. Further, the implementation(s) of the protocols can exist independent to that of their declaration.

At Uber Technologies Inc. (Uber), protocols have been used widely, resulting in more than 9000 protocols in a Swift mobile app code base of 1.45 MLoC (million lines of code). Interestingly, the protocols have primarily been used to facilitate testing. In this scenario a protocol declaration is associated with two implementations, one for the actual features of the app, and one mock implementation for testing purposes. The test code is protected under a compile-time macro, so when the production version of app is created, the mocking implementations are eliminated entirely, leaving only the feature-relevant implementation in the shipped binary. This ensures that testing-related code is not shipped and also the binary conforms to the over-the-air (OTA) size restrictions imposed by Apple's app store [AppleOTA 2019].

Extensive use of protocols introduces three types of performance overheads. First, a method invocation through a variable of protocol type requires indirection through a dispatch table, known in Swift as a *protocol witness table*. Such calls are analogous, e.g., to an interface call in Java [Ishizaki et al. 2000]. These indirect calls are costly at runtime and hinder other crucial optimizations like inlining. Second, the Swift type system distinguishes between value and reference types, and it also allows both value and reference types to implement a single protocol. The Swift compiler generates code for such cases via boxing and unboxing operations, enabling methods to operate generically over such protocols. These operations introduce both runtime overhead and some amount of code bloat. Finally, for each protocol, Swift generates global data structures like protocol witness tables, and as the number of protocols increase, the presence of these tables can impact app startup time.

In this paper, we describe the design and implementation of three compiler optimizations designed to eliminate much of this protocol-based overhead for common cases.[2] The optimizations stem from the well-known insight that if a protocol-typed variable can only have a single concrete type at runtime, the code can be rewritten to operate directly on that concrete type. The Swift compiler already contains optimizations that remove protocol-based overheads based on this insight for simple, local cases. Our techniques leverage more sophisticated intra-procedural value flow analysis and global analysis to optimize many more cases.

Within a procedure, when local variables are declared as protocol types but have a single concrete type, we eliminate the dispatch and boxing overheads for the protocol with an optimization named `LocalVar`. We employ a data flow analysis that tracks when there is an available variable that holds the contents of a boxed protocol, or "existential container" (see Section 2 for further background), and then employ a transformation that replaces uses of a container with that variable. This transformation eliminates both dispatch and boxing overheads from the protocol. The analysis precisely handles pointer-based operations used in boxing, and the transformation can introduce

[1] https://www.tensorflow.org/swift

[2] We also investigated optimizing via source-to-source transformation, simply replacing references to a protocol with the concrete implementation name for simple cases. But, several language features of Swift severely limited the applicability of this approach. For the large mobile application from Uber, we found several problematic cases, such as usage of the static meta-type of a protocol via `.self` and differences in access visibility between the protocol and its implementation.

new variables to enable further optimization in the presence of conditional updates (details in Section 4.2).

For cases where a callee has a protocol parameter but a call site passes a value with a specific concrete type, we employ a transformation `Param` to enable optimization. Such cases could be handled via inlining, but excessive inlining can lead to code bloat. `Param` instead rewrites the method signature to use a constrained generic type (details in Section 4.3), enabling Swift's existing generic specializer to remove the protocol overheads *without* requiring inlining of the callee.

Finally, when a global analysis identifies a single concrete type for some protocol variable, we employ a `SoleType` transformation to inject downcast instructions reflecting the concrete type information, which can then be leveraged for call site devirtualization. Currently our global analysis is a type hierarchy analysis [Dean et al. 1995], but the transformation is agnostic to the type of analysis used. `SoleType` interacts fruitfully with the other optimizations—its injected downcasts can be used to guide the specialization of `Param`, and it can leverage information from `LocalVar` to discover better locations for injecting casts, enabling further optimization.

From a program analysis perspective, the techniques we employ for our optimizations (class hierarchy analysis, intra-procedural dataflow analysis) are standard. The novelty of the present work is in the application of these analyses to optimizing Swift protocols, which pose unique challenges that were not faced by many previous works, like elimination of boxing overheads. Our solutions are also carefully designed for practical usage in a real-world compiler. Rather than performing a full-blown points-to analysis, we develop a dataflow analysis that surgically removes protocol overheads by only tracking the flow of concrete values through protocol variables; rather than performing whole-program class-hierarchy analysis, we perform module-level protocol conformance analysis; and finally, rather than performing program-wide generic specialization, we perform specialization only at the call sites when the concrete type is known so as to reap the maximum benefit without incurring a very high analysis cost.

We have implemented the aforementioned optimizations on top of the Swift compiler v5.0.0. We have evaluated the usefulness of these optimizations by applying them on 13 benchmarks from Swift performance benchmark suite and a `BucketSort` algorithm from BucketSort [2019]. On average, we observe a 1.56× speedup by executing the binary obtained by applying all the three optimizations compared to the baseline using optimization flag "-O -wmo". Further, our experimental results demonstrate a reduction in code size as well. More importantly, we have also evaluated our optimizations on a very large proprietary mobile app from Uber consisting of close to 1.5 million lines of Swift code and have observed speedups ranging from 6.9% to 55.49% on key spans in the app identified by other developers.

1.1 Technical Contributions

We make the following technical contributions in this paper:

- We describe the runtime and code size inefficiencies due to the wide spread use of protocols in Swift code bases.
- We give a partial semantics of protocol-related instructions in the Swift Intermediate Language (SIL).
- We present the design of three compiler optimizations targeted towards reducing the inefficiencies of protocols.
- We evaluate our optimizations on a number of benchmarks from Swift Performance Benchmark suite. We demonstrate an average speedup of 1.56× compared to the baseline. We also show performance benefits for a number of core functionalities in a large proprietary Swift application.

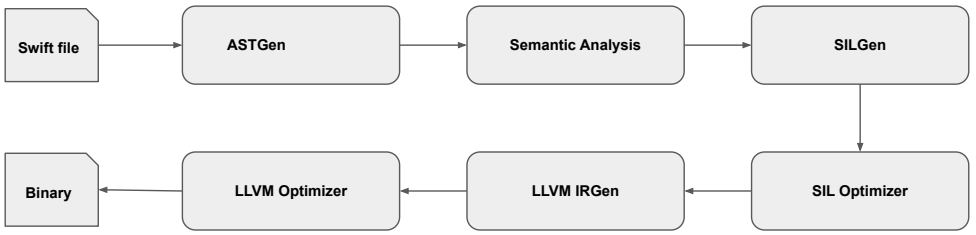

Fig. 1. Swift compiler passes.

2 SWIFT BACKGROUND

A *protocol* in Swift defines a set of requirements on methods, properties, and associated types. A type *conforms* to a protocol if its implementation satisfies the protocol's requirements and it declares the conformance. Both reference types (classes) and value types (e.g., structs and enums) can conform to protocols. A *class protocol* can only be conformed to by class types, whereas a *non-class protocol* can be conformed to by either classes or value types. Multiple protocols can be combined into a single requirement with protocol composition. Swift provides access modifiers such as `open`, `public`, `internal`, `fileprivate`, and `private` for assigning specific access controls for properties and methods of a protocol. Open access is the least restrictive access level where as private access is the most restrictive access level.

Figure 1 depicts the overall flow of Swift compiler. As a first step, the input Swift file is parsed to produce an AST on which various semantic checks are performed. Subsequently, SILGen generates Swift Intermediate Language (SIL) from the AST and performs various inter- and intra-procedural optimizations. Later on, LLVM IR is generated from SIL, further optimized, and an executable binary is finally produced. Swift supports both single-file and whole-module compilation (via compiler flag *-wmo*) modes. With whole-module compilation, all the input files of a single Swift module are optimized together at the SIL level. This enables interprocedural optimizations across files within the same module, particularly for methods and properties that are accessed with access levels `internal` or `private` (as they are not visible outside the module).

At the SIL level, protocol types are implemented using *Existential Containers* (EC), a boxed representation. (Protocol types are a form of existential types; see Chapter 24 of Pierce [2002].) An existential container for protocol P holds a reference to a value of some conforming type T and a reference to a *Protocol Witness Table* (PWT) showing which methods of T implement the requirements of P. An EC for a class protocol is referred to as a "class existential container" (or simply "class container"). Class containers are created and manipulated using the init_existential_ref and open_existential_ref SIL instructions. An EC for a non-class protocol is referred to as an "opaque existential container" (or simply "opaque container"). Opaque containers are manipulated using SIL init_existential_addr, open_existential_addr and deinit_existential_addr instructions. Furthermore, copy_addr and destroy_addr operations copy and destroy the contained concrete value in the opaque container. Further details on these existential operations are provided in Section 4.

Existential containers introduce additional overheads when lowered from SIL to LLVM IR. The compiler uses a special storage layout for ECs consisting of a stack allocated three-word *value buffer*, a *Value Witness Table* (VWT), and a protocol witness table. If the conforming type is a reference type, the reference itself is stored in the value buffer. If it is a value type and it fits in the three word value buffer, it is stored directly, and otherwise it is stored on the heap and a reference is

```
1  protocol Sorter {
2      func sort<T: Sortable>(items: [T]) -> [T]
3  }
4
5  struct InsertionSorter: Sorter {
6      func sort<T: Sortable>(items: [T]) -> [T] { /* implementation elided */}
7  }
8
9  struct Bucket<T: Sortable> {
10     var elements: [T]
11     func sort(algorithm: Sorter) -> [T] {
12         return algorithm.sort(elements)
13     }
14 }
15
16 func bucketSort<T>(elements: [T], sorter: Sorter, buckets: [Bucket<T>]) -> [T] {
17     // distribute elements contained in buckets, storing result in bucketsCopy (elided)
18     ...
19     for bucket in bucketsCopy {
20        results += bucket.sort(sorter) // results is array of T
21     }
22     return results
23 }
24
25 var array: [Int] = ... // array of integers
26 var buckets = [Bucket<Int>]() // Array of buckets of integers
27 let sorted = bucketSort(array, sorter: InsertionSorter(), buckets: buckets)
```

Listing 1. Motivating Example

stored in the value buffer. The VWT is also created for a value type in order to manage the lifetime of a value, including heap-allocated large values. A VWT exposes additional APIs for allocation and deallocation: *allocate*, *copy*, *destruct*, and *deallocate*. Additional details on the EC layout can be found in the presentation [WWDCSwiftPerf 2016].

The data structure for an EC and the API calls required to manipulate its content add significant boxing and unboxing overheads. Additionally, method invocations through protocol types are dynamically dispatched using the witness_method SIL instruction and cannot be inlined. Moreover, if a Swift application uses a large number of protocols, it can increase overall startup time due to the cost of loading the global data structures related to protocol and value witness tables.

The optimizations described in this paper target eliminating these overheads. Such overheads were particularly acute in the Uber code base due to heavy use of protocols to support mock objects for testability, as discussed in Section 1. In production builds of this code base, there are many protocols with exactly one implementation (as the mock implementations are not present), providing many opportunities for optimization.

3 MOTIVATION

In this section, we provide examples to motivate our optimizations of protocol overheads and to give an overview of how the optimizations work.

3.1 Existential Parameter Specializer

Listing 1 shows a bucket sorting algorithm adapted from BucketSort [2019] that uses Swift protocols. The protocol `Sorter` defines a function `sort` that operates over elements of generic type `T` (lines 1-3). `T` is constrained to conform to protocol `Sortable` (not shown) to ensure a comparison operation is present for sorting. `InsertionSorter` is one implementation of `Sorter`, which implements an

```
1  struct InsertionSorter {
2      func sort(items: [Int]) -> [Int] { ... }   // func is specialized to Int
3  }
4  // array and function are specialized to Int
5  struct Bucket {
6      var elements: [Int]
7      func sort() -> [Int] {
8          // syntactic sugar to show the absence of witness table lookup
9          return sort@line2(elements)
10     }
11 }
12 func bucketSort(elements: [Int], buckets: [Bucket<Int>]) -> [Int] {
13     ...
14     for bucket in bucketsCopy {
15         results += bucket.sort() // the sorter parameter is no longer necessary
16     }
17     return results
18 }
19 ...
20 let sorted = bucketSort(array, buckets: buckets) // the sorter param is removed
```

Listing 2. Optimized version of the example shown in Listing 1

insertion sort (lines 5-7). Elements within a bucket are sorted using a parameter `algorithm` of the `Sorter` protocol (lines 9-14).

For a given `array` of elements of type `Int`, an array of buckets of integers is defined (lines 25–26). `bucketSort` takes the `array`, the underlying sorting implementation employed within a bucket (`InsertionSorter`), and the array of buckets as input (line 27). The actual implementation distributes the elements across various buckets (not shown in the listing). Subsequently, sorting on each bucket is undertaken by using the provided `Sorter` implementation. A concatenation of the sorted elements within the buckets is returned, which corresponds to the sorted order for the input `array` (lines 19–22).

While Listing 1 provides a generic implementation that can be used to sort elements of *any* type, it imposes non-trivial overhead compared to an implementation specialized to a particular element type and sorting algorithm. The main overhead is the implementation of the sorting algorithm, which must perform comparison operations through indirect calls on values of `Sortable` type. Swift has extensive support for specializing generic methods based on the types present at a call site. However, in Listing 1 the sorting routine itself is invoked through an indirect call at line 12 (via protocol witness method lookup), thwarting generic specialization.

Listing 2 gives a source-level view of the optimizations enabled by our technique. The key protocol-related optimization is at line 9: rather than invoking a sort routine via a protocol witness table lookup, our optimizations enable rewriting this call to directly invoke the `InsertionSorter.sort` method. With this call devirtualized, the Swift compiler can create a specialized copy of `InsertionSorter.sort` that operates directly on `Int` values, rather than using indirect `Sortable` operations. This optimized implementation significantly outperforms the generic implementation shown in Listing 1, achieving a speedup of 21.75× for an array size of 10 million random elements.

For this example, our technique optimizes away the protocol dispatch by introducing new generic type parameters in a targeted manner. Observing that an `InsertionSorter` value is passed on line 27 of Listing 1, the optimizer rewrites the `bucketSort` signature as follows:

```
func bucketSort<T, U: Sorter>(elements: [T], sorter: U, ...
```

```
1  protocol P {
2   func foo() -> Int
3  }
4
5  class A : P {
6   func foo() -> Int { return 10 }
7  }
8
9  func bar(y:Bool) {
10  let a1:P  = A()
11  var a2:P  = A()
12  if y { a2 = a1 }
13  a2.foo()
14 }
15
16 func baz(a:P) { a.foo() }
```

Listing 3. Illustrative example.

```
17 func bar(y:Bool) {
18  x1 = alloc_ref A
19  x2 = alloc_ref A
20  x3 = x2
21  if y { x4 = x1 }
22  x5 = ϕ(x4, x3)
23  f = foo@line6
24  apply f(x5)
25 }
26
27 func baz(a:P) {
28  a1 = open_existential_addr a to
        opened_k2(P)
29  a2 = unchecked_addr_cast a1, A*
30  f = foo@line6
31  apply f(a2)
32 }
```

Listing 4. Optimized version of illustrative example.

```
33 func bar(y:Bool) {
34  // let a1:P  = A()
35  x1 = alloc_ref A
36  a11 = alloc_stack P
37  a12 = init_existential_addr a11 : A, P
38  store x1, a12
39
40  // var a2:P  = A()
41  x2 = alloc_ref A
42  a21 = alloc_stack P
43  a22 = init_existential_addr a21 : A, P
44  store x2, a22
45
46  if y {
47    // a2 = a1
48    copy_addr a11, a21
49  }
50
51  // a2.foo()
52  a24 = open_existential_addr a21 to
        opened_k1(P)
53  f = witness_method opened_k1(P) foo
54  apply f(a24)
55 }
56
57 func baz(a:P) {
58  // a.foo()
59  a1 = open_existential_addr a to
        opened_k2(P)
60  f = witness_method opened_k2(P) foo
61  apply f(a1)
62 }
```

Listing 5. SIL instructions for illustrative example.

Given this new signature, Swift's existing generic specializer creates a version of `bucketSort` specialized for an `InsertionSorter`. A similar transformation is performed to enable the key optimization of the `Bucket.sort` method. We describe the optimization in further detail in Section 4.3.

3.2 Eliminating Unnecessary Existential Containers

We now illustrate the additional costs imposed by use of existential containers, and how our optimizations can reduce these costs. Listing 3 presents a (contrived) code example designed to explain our techniques. It contains the definition of protocol `P`, a class `A` that conforms to `P`, and two methods `bar` and `baz` manipulating `P` and `A` variables.

Listing 5 sketches the Swift Intermediate Language (SIL) representation generated by the compiler for Listing 3. (A more detailed semantics for the instructions is given in Section 4.) Here, we observe instruction bloat for existential-related operations. For instance, each definition of a protocol variable (line 34) corresponds to four SIL instructions (lines 35-38) that construct the corresponding existential container. The boxing operation includes allocating a reference for an object of type `A`, allocating stack space for the container, initializing the container, and finally storing the object reference into the container. Any use of the contained values requires "unboxing" instructions, e.g., line 52. Additionally, the invocations of `foo` involves witness table lookups (lines 53 and 60).

$$
\begin{aligned}
\textbf{instructions } i ::= {} & x_i = \text{alloc\_stack } T \\
| \; & \text{store } x_i \textit{ to } x_j \\
| \; & \text{copy\_addr } x_i \textit{ to } x_j \\
| \; & x_j = \text{init\_existential\_ref } x_i : C, P \\
| \; & x_j = \text{open\_existential\_ref } x_i \text{ to } \textit{opened}_k(P) \\
| \; & x_j = \text{init\_existential\_addr } x_i : T, P \\
| \; & x_j = \text{open\_existential\_addr } x_i \text{ to } \textit{opened}_k(P) \\
| \; & x_i = \text{witness\_method } \textit{opened}_k(P) \textit{ sig}
\end{aligned}
$$

Fig. 2. SIL instructions relevant to our analysis and optimization.

A version of the Listing 5 optimized by our techniques appears in Listing 4. In this version, the existential containers have been completely optimized away. Note that to do so, the optimizer has introduced new SIL variables and a phi statement, to track the data flow of the values previously stored in the existential containers. This transformation requires careful tracking of variables corresponding to containers, particularly in the presence of address-based manipulations like lines 37–38 and 48. This `LocalVar` optimization is described in detail in Section 4.2.

Our `SoleType` transformation injects downcasts of values in existential containers when some other analysis discovers a single concrete type for the values. For example, consider the cast instruction at line 29. For this example, a type hierarchy for the full program shows that `P` has only one conforming implementation, `A`, so such a cast is safe. The cast enables devirtualization of the subsequent `foo` call. In general, `SoleType` is agnostic to which other analysis provides the precise type information, and it could be integrated with global pointer analysis or even profile-guided techniques.

The key sophistication in `SoleType` is its fruitful interactions with `LocalVar`. Given results from `LocalVar`'s data flow analysis, `SoleType` can choose program points early in methods to inject casts. In turn, `LocalVar` can be re-run with the cast present and leverage it to remove more existential containers. `SoleType` is described in detail in Section 4.4.

4 EXISTENTIAL SPECIALIZATION

In this section we present our optimizations in full technical detail. We start by giving a (partial) semantics for the relevant SIL instructions in Section 4.1. We then describe the `LocalVar` analysis and transformation used for eliminating existential containers in Section 4.2. Section 4.3 describes the `Param` transformation for specializing protocol-typed parameters, and Section 4.4 presents the `SoleType` transformation for injecting casts with additional type information. Finally, Section 4.5 describes enhancements to dead-code elimination motivated by the previous optimizations.

4.1 Semantics

Figure 2 lists the SIL instructions we model for the purposes of our optimization. We include the alloc_stack, store, and copy_addr instructions for allocating and modifying memory. The init_existential_ref and open_existential_ref instructions create and open class containers for reference types (see Section 2), while the init_existential_addr and open_existential_addr instructions handle opaque containers, which can contain either a value or a reference. The witness_method instruction performs a method lookup on a protocol type.

A method invocation on a value in an existential container proceeds via the following instruction sequence:

(1) The container is opened using open_existential_ref or open_existential_addr as appropriate;
(2) a witness_method instruction performs the method lookup; and finally
(3) an apply instruction invokes the method with appropriate parameters.

For method invocations, our optimizations primarily modify method lookup, but not the actual invocation.[3] So, we elide modeling of method invocations and the apply instruction here.[4]

SIL uses *archetypes* to connect the protocol witness table referenced by an existential container to the method lookup performed by a corresponding witness_method instruction. Each open_existential_ref and open_existential_addr instruction is tagged with a unique archetype id, naming the "opened" existential type of the instruction's return value. A subsequent witness_method instruction references an archetype id when performing a method lookup, connecting it to a corresponding open instruction. Using these ids, the compiler determines which protocol witness table to use for a method lookup. Archetypes help to simplify SIL by enabling use of the same witness_method lookup instruction across all types of existential containers.[5]

Figure 3 gives a *partial* semantics for our modeled SIL instructions, capturing the behaviors relevant to our optimizations. A program state consists of an environment ρ mapping variables to values (where values include addresses), a store σ mapping addresses to values, and an archetype map α mapping archetype IDs to protocol witness tables. Note that in the Swift compiler, α is computed and used only at compile time; our modeling here shows the tracking that the compiler must do. The partial semantics elides various safety checks like ensuring well-typedness of instruction arguments or that memory is initialized before use; the Swift compiler only generates well-formed SIL code that respects these properties (modulo bugs).

The rules in Figure 3 show how each SIL instruction transitions an input state ρ, σ, α to an updated state ρ', σ', α'. An alloc_stack instruction allocates a fresh memory location a and marks it as uninitialized via a special `uninit` value. (We discuss stack vs. heap allocation in Section 4.1.1.) A store instruction writes the value in variable x_i to the memory address referenced by x_j. copy_addr copies a value from the address in x_i to the address in x_j.

An init_existential_ref instruction takes as arguments a variable x_i holding the address a of an object o, the class C of o, and a protocol P that C conforms to. It finds the protocol witness table K indicating how methods in P should be dispatched on a C object, and stores K and a in a new class existential container $clContainer(K, a)$.[6] An open_existential_ref instruction takes a variable x_i holding a class existential container $clContainer(K, a)$ as an argument. The instruction is labeled with some archetype type $opened_k(P)$, where k is unique to the instruction. The instruction both extracts the contained address a and updates the archetype map α to map id k to the contained protocol witness table K.

Allocation and initialization of opaque existential containers require an instruction sequence similar to the following (; is a sequencing operator):

$$x_k = \ldots;\ x_i = \text{alloc\_stack } P;\ x_j = \text{init\_existential\_addr } x_i : T, P;\ \text{store } x_k \text{ to } x_j$$

The memory for the opaque container is always stack-allocated (see Section 4.1.1) using an alloc_stack instruction. The address a of this memory is then passed to an init_existential_addr instruction, along with a concrete type T and corresponding protocol P. init_existential_addr stores an opaque existential container $opContainer(K, a')$ into a, with $K = pwt(T, P)$ (similar to the init_existential_ref case) and a' for storing the contained value. a' may be an address within the block of memory allocated by alloc_stack (for reference types and inline storage of small values)

[3] Invocations are affected only in that the calling convention is modified after devirtualization.
[4] The `Param` optimization in Section 4.3 *inspects* apply instructions, but it does not transform them.
[5] SIL has five types of existential containers [SILDocs 2019]; we only describe the two most common ones here for simplicity.
[6] *clContainer* and *opContainer* construct SIL-level data types, not accessible to Swift source.

$$\boxed{\rho, \sigma, \alpha, s \rightarrow \rho', \sigma', \alpha'}$$

$$\textsc{AllocStack}\ \frac{a \text{ is fresh}}{\rho, \sigma, \alpha, x_i = \texttt{alloc\_stack}\ T \rightarrow \rho\left[x_i \mapsto a\right], \sigma\left[a \mapsto \texttt{uninit}\right], \alpha}$$

$$\textsc{Store}\ \frac{\rho(x_i) = v \qquad \rho(x_j) = a}{\rho, \sigma, \alpha, \texttt{store}\ x_i\ \textit{to}\ x_j \rightarrow \rho, \sigma\left[a \mapsto v\right], \alpha}$$

$$\textsc{CopyAddr}\ \frac{\rho(x_i) = a_1 \qquad \sigma(a_1) = v \qquad \rho(x_j) = a_2}{\rho, \sigma, \alpha, \texttt{copy\_addr}\ x_i\ \textit{to}\ x_j \rightarrow \rho, \sigma\left[a_2 \mapsto v\right], \alpha}$$

$$\textsc{InitExRef}\ \frac{\rho(x_i) = a \qquad K = pwt(C, P) \qquad c = clContainer(K, a)}{\rho, \sigma, \alpha, x_j = \texttt{init\_existential\_ref}\ x_i : C, P \rightarrow \rho\left[x_j \mapsto c\right], \sigma, \alpha}$$

$$\textsc{OpenExRef}\ \frac{\rho(x_i) = clContainer(K, a)}{\rho, \sigma, \alpha, x_j = \texttt{open\_existential\_ref}\ x_i\ \texttt{to}\ opened_k(P) \rightarrow \rho\left[x_j \mapsto a\right], \sigma, \alpha\left[k \mapsto K\right]}$$

$$\textsc{InitExAddr}\ \frac{\rho(x_i) = a \qquad K = pwt(T, P) \qquad a' = valAddr(a, T, P) \qquad c = opContainer(K, a')}{\rho, \sigma, \alpha, x_j = \texttt{init\_existential\_addr}\ x_i : T, P \rightarrow \rho\left[x_j \mapsto a'\right], \sigma\left[a \mapsto c\right]\left[a' \mapsto \texttt{uninit}\right], \alpha}$$

$$\textsc{OpenExAddr}\ \frac{\rho(x_i) = a \qquad \sigma(a) = opContainer(K, a')}{\rho, \sigma, \alpha, x_j = \texttt{open\_existential\_addr}\ x_i\ \texttt{to}\ opened_k(P) \rightarrow \rho\left[x_j \mapsto a'\right], \sigma, \alpha\left[k \mapsto K\right]}$$

$$\textsc{WitMeth}\ \frac{\alpha(k) = K \qquad K(sig) = m}{\rho, \sigma, \alpha, x_i = \texttt{witness\_method}\ opened_k(P)\ sig \rightarrow \rho\left[x_i \mapsto m\right], \sigma, \alpha}$$

Fig. 3. Partial semantics for relevant SIL instructions.

or for newly-allocated memory (for storing large values) [WWDCSwiftPerf 2016]; the *valAddr* function abstracts this logic. To complete initialization, the appropriate T value is written to a' via a store or copy_addr instruction.

Finally, the witness_method instruction performs a method lookup. It takes as arguments an archetype type $opened_k(P)$ and a method signature *sig*. It looks up the archetype id k in α to get the corresponding protocol witness table K, and uses K to find the appropriate method m based on *sig*.

4.1.1 Key Invariants. The soundness of our optimizations rely on the following key invariants regarding existential containers in SIL:

- Class existential containers (created by init_existential_ref) are of loadable types with SSA representation and hence are immutable, i.e., the contained reference never changes after initialization. store and copy_addr instructions only update opaque containers (but not class containers) that are of address types, which are guaranteed by `checkStoreInst` and `checkCopyAddrInst` in SILVerifier [SILVerifier 2017], respectively.

- An opaque container *must* be stack allocated if it is the operand of an init_existential_addr instruction or the target operand of a store or copy_addr instruction. The opaque container is always copied before reading from / storing to the heap.
- At any point during program execution, there is at most one location in stack memory corresponding to an alloc_stack instruction. In particular, SIL enforces a discipline that ensures that any memory stack-allocated in a loop does not outlive the current loop iteration.

These invariants are guaranteed by the current frontend of Apple's Swift compiler. Our data flow analysis relies on these invariants to perform strong updates on (abstract) containers for store and copy_addr instructions, as detailed in the next section.

4.2 Flow-Based Optimization

Given the semantics in Section 4.1, we now describe `LocalVar`, our core flow analysis and optimization for devirtualizing witness method calls and removing unnecessary existential containers. `LocalVar` is able to completely remove an existential container c in cases where it can determine both the concrete type of the value v in c *and* another SIL variable guaranteed to already hold v. `LocalVar` first uses dataflow analysis to discover existential containers for which there is a corresponding SIL variable (or variables) holding their contained value. It then performs a transformation to replace uses of such containers, after which dead code elimination can remove the containers entirely. We first describe the dataflow analysis, and then present the subsequent code transformation.

4.2.1 Flow Analysis. The abstract domain for our flow analysis consists of pairs of maps $\langle \Gamma, \Sigma \rangle$. Γ maps each SIL variable to a set of *abstract containers* it may point to (i.e., contain the address of). Abstract containers are static representations of runtime existential containers. We use an allocation site abstraction, representing containers allocated by all executions of an instruction with a single abstract container. However, note that by the invariants given in Section 4.1.1, we know that for opaque containers used in init_existential_addr, store, and copy_addr instructions, at most one concrete container can be alive at a time for the corresponding abstract container. We assume a unique identifier l for each alloc_stack and init_existential_ref instruction, and write c_l for the abstract container representing the allocations by instruction l.

Σ maps each abstract container to the set of all SIL variables possibly holding its contained value. SIL employs static single assignment (SSA) form, so each SIL variable has a unique defining instruction. Hence, given Σ, we can find both the instructions creating the value stored in an existential container and the possible concrete types of the value. We leverage a special marker variable $x_\top$ to handle cases where the analysis cannot precisely track the variables holding a container's value, e.g., when the container is passed in as a method parameter.

At the start of analysis, Γ and Σ at each program point respectively map each SIL variable and abstract container to $\emptyset$. Our lattice is a standard subset lattice extended to pairs of maps:

$$\langle \Gamma, \Sigma \rangle \sqsubseteq \langle \Gamma', \Sigma' \rangle \equiv (\forall x_i.\ \Gamma(x_i) \subseteq \Gamma'(x_i)) \wedge (\forall c_i.\ \Sigma(c_i) \subseteq \Sigma'(c_i))$$

And at control-flow merge points, we apply the following join function $\sqcup$:

$$\langle \Gamma, \Sigma \rangle \sqcup \langle \Gamma', \Sigma' \rangle \equiv \langle \Gamma \uplus \Gamma', \Sigma \uplus \Sigma' \rangle$$

Here, $\uplus$ is a map union operator that combines values for the same key using $\cup$.

Table 1 gives the transfer functions for our dataflow analysis. For an init_existential_ref instruction at line l, we update Γ to map the assigned variable x_j to the corresponding abstract container c_l, and update Σ to map c_l to the argument variable x_i. Handling opaque existential containers is more complex due to their multi-instruction initialization sequence (see Section 4.1). For an

Table 1. Transfer functions for dataflow analysis.

Stmt	Result State
$l : x_j = \text{init\_existential\_ref } x_i : C, P$	$\langle \Gamma \left[x_j \mapsto \{c_l\}\right], \Sigma \left[c_l \mapsto \{x_i\}\right] \rangle$
$l : x_i = \text{alloc\_stack } P$	$\langle \Gamma \left[x_i \mapsto \{c_l\}\right], \Sigma \rangle$
$x_j = \text{init\_existential\_addr } x_i : T, P$	$\langle \Gamma \left[x_j \mapsto \Gamma(x_i)\right], \Sigma \rangle$
store x_i *to* x_j, $\Gamma(x_j) = \{c_l\}$	$\langle \Gamma, \Sigma \left[c_l \mapsto \{x_i\}\right] \rangle$
store x_i *to* x_j, $\lvert\Gamma(x_j)\rvert > 1$	$\langle \Gamma, \mathit{addTop}(\Sigma, \Gamma(x_j)) \rangle$
copy_addr x_i *to* x_j, $\Gamma(x_i) = \{c_l\}, \Gamma(x_j) = \{c_{l'}\}$	$\langle \Gamma, \Sigma \left[c_{l'} \mapsto \Sigma(c_l)\right] \rangle$
copy_addr x_i *to* x_j, $\lvert\Gamma(x_i)\rvert > 1 \vee \lvert\Gamma(x_j)\rvert > 1$	$\langle \Gamma, \mathit{addTop}(\Sigma, \Gamma(x_j)) \rangle$
x_i leaked, $\Gamma(x_i) \neq \emptyset$	$\langle \Gamma, \mathit{addTop}(\Sigma, \Gamma(x_i)) \rangle$
$x_k = \phi(x_i, x_j)$	$\langle \Gamma \left[x_k \mapsto \Gamma(x_i) \cup \Gamma(x_j)\right], \Sigma \rangle$

alloc_stack instruction at line l, we update Γ for the corresponding c_l; our implementation only updates the abstract state when P is a protocol type.

The init_existential_addr transfer function is the most counter-intuitive. The function updates Γ to map the assigned variable x_j to the same container(s) mapped by the input variable x_i. But, init_existential_addr actually returns an address for the contained value, *not* for the container (see Figure 3). So how can this rule work? The key insight is that our analysis is concerned only with the values stored in existential containers, *not* the protocol witness tables. Hence, the rule treats the container as a direct box for the contained value and ignores the protocol witness table initialization performed by init_existential_addr, instead treating the instruction like a copy.

store and copy_addr instructions are modeled with *strong updates* when possible. For store, we have two cases. In the case where our analysis shows the destination variable x_j can only point to abstract container c_l, we update Σ to map c_l to $\{x_i\}$. This treatment is sound since we know in this case that c_l corresponds to exactly one concrete opaque container (see Section 4.1.1). When $|\Gamma(x_j)| > 1$, the analysis updates Σ to add $x_\top$ to the variable set for all abstract containers in $\Gamma(x_j)$, as the analysis cannot determine which container is getting updated. For this case we leverage the auxiliary function *addTop*:

$$\mathit{addTop}(\Sigma, C)(c) = \begin{cases} \Sigma(c) \cup x_\top & c \in C \\ \Sigma(c) & \text{otherwise} \end{cases}$$

Similarly, the analysis performs a strong update for copy_addr instructions when both the source and target variable each point to exactly one abstract container, and conservatively adds $x_\top$ otherwise.

If any variable x_i containing an abstract container is leaked out of the function scope (e.g., by being passed to a callee via an **inout** parameter), the analysis adds $x_\top$ to the abstract state for the corresponding container(s), as the analysis cannot ensure that the containers do not get updated by other code. Finally, ϕ statements are handled by taking the union of abstract container sets for the argument variables. Note that this ϕ handling is in addition to applying the join function $\sqcup$ defined earlier at control-flow merge points. $\sqcup$ is needed to merge states for abstract containers in Σ, as their updates via store and copy_addr instructions are not reflected in ϕ statements.

Note that our analysis solves a problem similar to intra-procedural must-alias analysis for single-level pointers, as described by Landi and Ryder [1991]. Our analysis need only track a single level of pointers for opaque containers, since only aliasing with variables holding the same value is relevant for our transformation. We do not directly adapt the algorithms of Landi and Ryder as we found our dataflow formulation to integrate more naturally with the subsequent transformations for optimization, described in Section 4.2.2.

Table 2. Partial results for dataflow analysis on Listing 5.

Line	State After
36	$\langle[a_{11} \mapsto \{c_{36}\}], []\rangle$
37	$\langle[a_{11} \mapsto \{c_{36}\}, a_{12} \mapsto \{c_{36}\}], []\rangle$
38	$\langle[a_{11} \mapsto \{c_{36}\}, a_{12} \mapsto \{c_{36}\}], [c_{36} \mapsto \{x_1\}]\rangle$
44	$\langle[a_{11} \mapsto \{c_{36}\}, a_{12} \mapsto \{c_{36}\}, a_{21} \mapsto \{c_{42}\}, a_{22} \mapsto \{c_{42}\}], [c_{36} \mapsto \{x_1\}, c_{42} \mapsto \{x_2\}]\rangle$
48	$\langle[a_{11} \mapsto \{c_{36}\}, a_{12} \mapsto \{c_{36}\}, a_{21} \mapsto \{c_{42}\}, a_{22} \mapsto \{c_{42}\}], [c_{36} \mapsto \{x_1\}, c_{42} \mapsto \{x_1\}]\rangle$
50	$\langle[a_{11} \mapsto \{c_{36}\}, a_{12} \mapsto \{c_{36}\}, a_{21} \mapsto \{c_{42}\}, a_{22} \mapsto \{c_{42}\}], [c_{36} \mapsto \{x_1\}, c_{42} \mapsto \{x_1, x_2\}]\rangle$

Example. Table 2 shows partial results for running the dataflow analysis on the example in Listing 5. For each line number, the table shows the entries in Γ and Σ after the line (excluding entries with value $\emptyset$). The first three rows show handling of the container initialization in lines 36–38. After line 38, the analysis concludes that variable `x1` holds the value stored in the container allocated on line 36. Similarly, after line 44, the analysis knows that `x2` holds the value stored in container c_{42}. The copy_addr at Line 48 overwrites the contents of c_{42}, the container referenced by a_{21}. Here, the analysis is able to do a strong update and conclude that after the instruction, `x1` holds the value in c_{42}. Finally, line 50 is a control-flow merge, so the analysis computes a join of the states from lines 44 and 48, and concludes that either `x1` or `x2` may hold the value in c_{42}.

4.2.2 Transformation. Given the results of the flow analysis, our transformation pass proceeds as follows. Recall the standard sequence of instructions used for a witness method invocation (discussed in Section 4.1):

$$x_j = \text{open\_existential\_addr } x_i \text{ to } \mathit{opened}_k(P);\ x_f = \text{witness\_method } \mathit{opened}_k(P)\ \mathit{sig};\ \text{apply } x_f(x_j, \ldots)$$

We identify open_existential_ref and open_existential_addr instructions in such sequences where for the argument variable x_i, $\Gamma(x_i) = \{c_l\}$, $\Sigma(c_l) \neq \emptyset$, and $x_\top \notin \Sigma(c_l)$. In this case, $\Sigma(c_l)$ contains the variables possibly holding the value stored in the container c_l. If $\Sigma(c_l)$ is a singleton $\{x_m\}$, we can simply remove the open instruction,[7] replace the subsequent witness_method instruction with a function_ref instruction referencing the method to invoke,[8] and update the invocation apply instruction to pass x_m directly, i.e.: $x_f = \text{function\_ref } \mathit{full\_sig};\ \text{apply } x_f(x_m, \ldots)$. Here, *full_sig* is the resolved signature of the invoked function based on the concrete type.

$\Sigma(c_l)$ can contain more than one variable in the case of control-flow merges. But, in some cases, we can still transform away the open instruction by introducing new SIL variables and ϕ statements at the control-flow merge points where the incoming states for c_l differed. Once these ϕ statements are introduced, there will be a new SIL variable x_m corresponding to the latest merge point, and we can use x_m to perform the desired replacement. For the Listing 5 example, at line 52 either `x1` or `x2` may hold the value in the container referenced by `a21` (see final line of Table 2). For this case, the transformation introduces a statement `x5 = ϕ(x4, x3)` at the previous control-flow merge, with corresponding definitions for `x4` and `x3`, as shown in Listing 4. Then, the call can be rewritten to directly invoke `A.foo()`, passing `x5`.

The above discussion assumes that once we have the variable x_i holding the value boxed by an existential container, we can immediately determine the concrete type of x_i by looking at its definition. However, in some cases the defining instruction may itself be an open_existential_addr or open_existential_ref instruction, corresponding to another container. Our implementation tracks data flow recursively through these containers to compute a final set of relevant variables.

[7] Our implementation actually leverages an existing Swift optimization to remove the open instruction by introducing a new alloc_stack instruction.

[8] A SIL function_ref instruction directly loads the method referenced by its signature argument.

```
63 func fizz() {
64   let x1: A = A()
65   var p: P = x1
66   for i in 1...2 {
67     let x2: A = A()
68     let p2: P = x2
69     if i == 1 {
70         p = p2
71     }
72     p.foo()
73   }
74 }
```

Listing 6. Example with loop.

```
75 func fizz() {
76  x1 = alloc_ref A
77  for i in 1..2 {
78    x4 = φ(x1, x5)
79    x2 = alloc_ref A
80    if i == 1 {
81      x3 = x2
82    }
83    x5 = φ(x4, x3)
84    f = foo@line6
85    apply f(x5)
86  }
87 }
```

Listing 7. Optimized version of loop example.

```
88 func fizz() {
89  x1 = alloc_ref A
90  p11 = alloc_stack P
91  p12 = init_existential_addr p11 : A, P
92  store x1, p12
93
94  for i in 1...2 {
95    x2 = alloc_ref A
96    p21 = alloc_stack P
97    p22 = init_existential_addr p21 : A, P
98    store x2, p22
99
100   if i == 1 {
101     copy_addr p21, p11
102   }
103
104   p3 = open_existential_addr p11 to
            opened_k1(P)
105   f = witness_method opened_k1(P) foo
106   apply f(p3)
107 }
```

Listing 8. SIL instructions for loop example.

Once we have replaced all instructions opening an abstract container c_l, it becomes useless, which should allow standard dead-code elimination to remove the init instruction for c_l. In practice we found that we had to enhance the existing Swift dead-code elimination pass to fully remove containers, as detailed in Section 4.5.

Example. Listing 6 gives a slightly more complex example code that includes a loop (the definitions of class `A` and protocol `P` are the same as in Listing 3).[9] Due to the loop, multiple objects allocated at line 67 and assigned to `x2` can be live simultaneously, so the transformation phase cannot naïvely use `x2` as a variable holding the container's value. We show here how our technique can soundly transform this example via additional ϕ statements to remove container overhead.

Listing 8 gives high-level SIL code for the example. At line 103, our flow analysis will determine that either `x1` or `x2` holds the value in container c_{90} allocated at line 90. Note that for this example, `p` will always be initialized at line 72 even *without* initializing `p` before the loop (as it gets set in the first iteration); one may wonder if our analysis would conclude (unsoundly) that `x2` always holds the value in c_{90} in such a case. However, Swift disallows such code, as it performs a path-*insensitive* analysis to determine if variables are initialized before they are used. To keep `p` uninitialized, the code must use an optional type `P?` that requires null checking before use.[10] Our optimizations do not transform code using optional types directly; they only apply once a value has been successfully read from an optional.

Listing 7 presents the optimized code for the example. The ϕ on line 83 is introduced for the container c_{90} at line 103 in Listing 8 to hold either `x1` or `x2`. Moreover, since the next iteration of the loop nest uses the result of this ϕ in line 83, the incremental SSA updater [SILSSA 2017] in Swift's SIL inserts another ϕ in the loop entry block at line 78 in order to maintain the SSA form of the

[9] This example is based on one provided by an anonymous reviewer.
[10] https://developer.apple.com/documentation/swift/optional

```
108 // non-class protocol NCP
109 protocol NCP  {...}
110 // SIL function foo taking p of type
        NCP as parameter.
111 sil func foo(p:NCP) {
112   /* use of p */
113 }
```

Listing 9. Non-Class Param Specializer Example (Before)

```
114 inline sil func foo(p:NCP) {
115   f = function_ref foo_inner
116   p1 = open_existential_addr p
117   apply f(p1)
118 }
119 sil func foo_inner <T1 where T1 : NCP>(
        p:T1) {
120   p1 = alloc_stack $NCP
121   p2 = init_existential_addr p1
122   copy_addr p to p1
123   destroy_addr p
124   /* use of p is replaced with p1 */
125   dealloc_stack p1
126 }
```

Listing 10. Non-Class Param Specializer Example (After)

intermediate representation. The concrete values, `x1` and `x2`, are subsequently used to devirtualize the `foo` method invocation on line 84. Moreover, the two containers, c_{90} and c_{96}, including other existential overheads are eliminated by the dead code elimination 4.5.

4.3 Existential Parameter Specialization (`Param`)

In some cases, the purely intra-procedural approach of Section 4.2 is insufficent for optimization. A common inter-procedural case occurs when a method formal parameter is of protocol type, but at a call site the concrete type of the actual parameter is evident, e.g., the `bucketSort` method and its call in Listing 1. Here, we describe our `Param` optimization, which leverages generic specialization to remove protocol overheads for this case *without* forcing inlining.

The central idea of `Param` is to rewrite the protocol type of the relevant parameter to instead use a fresh, constrained generic type parameter. E.g., given a function signature `foo(p: P)` where `P` is a protocol, `Param` effectively rewrites the signature to `foo<T: P>(p: T)`, where `T` is a fresh type variable. This alternate representation enables the Swift compiler's `GenericSpecializer` pass to create a specialized version of `foo` for some concrete type, removing the protocol overheads. `Param` only performs this transformation after determining that some specialization will occur, based on analyzing call sites. Also, note that if all call sites end up invoking a specialized version of the method, the original method will be deleted by dead-code elimination.

Our implementation differs slightly from the procedure described above. In actuality, `Param` outlines the body of the original method into a fresh generic method and modifies the original method body appropriately. The original method is also marked with the `inline` attribute to ensure it is always inlined. This technique allows `Param` to perform transformations locally on the target method, without having to modify all call sites of the method.

Listing 9 (before) and Listing 10 (after) depict how we optimize a non-class protocol parameter using `Param`. Lines 114-118 present the modified method with an `inline` attribute, calling the inner generic method on Line 117. Lines 119-126 show the body of the inner generic method. Lines 120-123 and 125 show how we box/unbox the generic parameter as an existential in order to reuse the old callee body. Similarly, Listing 11 (before) and Listing 12 (after) show how we transform class-protocol parameters.

In most cases in practice, the `Param` optimization is profitable in terms of both running time and code size. However, we note two caveats. First, if there are multiple concrete types that conform to a protocol and these concrete types are used in many different calling contexts, the specialization per concrete type may lead to code size bloat. Second, `Param` can introduce extra overhead if the transformed method stores the relevant parameter in a protocol-typed heap location. In this case,

```
127 // class protocol CP
128 protocol CP : class {...}
129 // SIL function foo taking p of type CP
        as parameter.
130 sil func foo(p:CP) {
131   /* use of p */
132 }
```

Listing 11. Class Param Specializer Example (Before)

```
133 inline sil func foo(p:CP) {
134   f = function_ref foo_inner
135   p1 = open_existental_ref p
136   apply f(p1)
137 }
138 sil func foo_inner <T1 where T1 : CP>(p
        :T1) {
139   p1 = init_existential_ref p
140   /* use of p is replaced with p1 */
141 }
```

Listing 12. Class Param Specializer Example (After)

while the original code may have been able to re-use an existential container, the transformed code may perform an additional unboxing and boxing operation. We observed this behavior in one benchmark and plan to enhance `Param` to avoid it in the future.

Returning to our motivating example from Listing 1, `Param` is the key optimization for this benchmark. Both the `bucketSort` and `Bucket.sort` routines are transformed by `Param`, enabling the de-virtualization of the call at line 12 that unlocks further specialization opportunities (see Section 3.1).

4.4 Sole Type Transformation

In certain cases, the local flow analysis of Section 4.2 cannot determine a single concrete type for a protocol-typed variable, but some other analysis like global type-hierarchy analysis is able to do so. In such cases, we perform a `SoleType` transformation, inserting a downcast to make the concrete type evident to other optimizations. In terms of SIL, we insert an unchecked_ref_cast or unchecked_addr_cast instruction to the concrete type, depending on whether we have a class or non-class protocol, respectively. By default, these casts are inserted immediately before call sites where the value is used. Then, further optimization can either devirtualize the call if the value is passed as the **`self`** argument, or specialize the callee using `Param` (see Section 4.3) if the value is passed in some other argument position.

When provided the analysis results of `LocalVar`, `SoleType` can compute a more optimal placement for cast instructions. Consider the example in Listing 13, and assume that field **`Struct`**.`somefield` is of protocol type `P`. The struct_element_addr instruction returns the address of `x0`.`somefield`, and the next two instructions copy the field's value into a new existential container. Then, in the loop, the container is opened twice for two different call sites. By default, `SoleType` would insert a downcast for each call site, enabling devirtualization but no other optimization of the local container. However, `LocalVar` determines that the container pointed to by `x2` holds the same value as container `x1`.[11] With this information, rather than inserting casts at the call sites, `SoleType` instead directly downcasts the value inside the container referenced by `x1`, by also injecting an open_existential_addr instruction. The final optimized code is shown in Listing 14. With this alternate cast, a subsequent run of `LocalVar` and dead code elimination can completely remove the local existential container (`x2`) and its overheads.

Currently, we perform a global type hierarchy analysis [Dean et al. 1995] to find cases where a protocol has a sole conforming type. This analysis is only performed when Swift's whole-module optimization mode is enabled. We can only conclusively analyze a protocol when its visibility is set to **`internal`** or stricter, as a public protocol may have conforming types in client modules. In

[11] The version of `LocalVar` formalized in Section 4.2 does not track information about such address variables, but our implementation does. The extension is straightforward and we elide it for the simplicity of presentation.

```
142  x1 = struct_element_addr x0 : $*Struct,
         #Struct.somefield
143  x2 = alloc_stack $P
144  copy_addr x1 to [initialization] x2 : $
       *P
145  Loop:
146    x3 = open_existential_addr
         immutable_access x2 : $*P to $*
         @opened P
147    apply(x3)
148    x4 = open_existential_addr
         immutable_access x2 : $*P to $*
         @opened P
149    apply(x4)
```

Listing 13. Combining LocalVar with SoleType (Before)

```
150  x1 = struct_element_addr x0 : $*Struct,
         #Struct.somefield
151  x2 = open_existential_addr
       immutable_access x1 : $*P to $*
       @opened P
152  x3 = unchecked_addr_cast x2 : $*opened
       P to $*K
153  Loop:
154    apply(x3)
155    apply(x3)
```

Listing 14. Combining LocalVar with SoleType (After)

the future, we plan to investigate using global pointer analysis to find more sole types, and also to use runtime profiles to find likely-sole types. For such cases, the SoleType transformation should work without modification.

4.5 Existential Dead Code Elimination

After implementing the aforementioned optimizations, we found that in some cases the Swift compiler's existing dead code elimination (DCE) could still not remove certain unnecessary existential containers, so we implemented a pass to perform some additional cleanup before the built-in DCE. The most important case to handle was an alloc_stack instruction allocating memory for a container, where the memory was initialized via a store instruction, but the container otherwise went unused (other than getting deallocated). For example, in Listing 5, after devirtualizing the call to `f` and passing the `A` parameter directly (see Listing 4), the alloc_stack at line 36 is unused, but the store at line 38 still initializes it. Built-in DCE is unable to clean up the existential container in such cases. Our extra pass removes the dead store instruction, and once removed, the built-in DCE can remove the alloc_stack instruction and corresponding de-allocation instructions. Via similar logic, our pass also removes copy_addr instructions that write into a container that is otherwise unused.

Note that the combination of our optimizations can lead to a protocol becoming completely unused. In such a case, the built-in DCE can remove the global protocol witness tables and value witness tables associated with the protocol. Our evaluation showed that on a real-world industrial code base, this cleanup could have a significant positive impact on app startup time.

5 IMPLEMENTATION

The overall framework for existential specializer is shown in Figure 4. Our optimization passes are integrated tightly into the SILOptimizer phase of the Swift compiler as shown in Figure 1. We have added two new analyses and four transformation passes apart from modifying the existing `SILCombiner` pass. The two new analyses are `SoleTypeAnalysis` and `LocalVarAnalysis`. The `SoleTypeAnalysis` attempts to find a sole type conforming to a protocol using type-hierarchy analysis [Dean et al. 1995]. `LocalVarAnalysis` performs the data-flow analysis described in Section 4.2 to determine the variable(s) holding the value in an existential container. If different variables hold the value on different control-flow paths, we transform the program using `LocalVarSSAUpdater` to insert additional ϕ instructions on the underlying variables, as described in Section 4.2.

The `Param` transformation described in Section 4.3 consumes the analysis results of `SoleTypeAnalysis` and `LocalVarAnalysis` as well as the existing `SingleWriteAnalysis` pass in the Swift compiler to decide when to introduce a constrained-generic type variable for a protocol

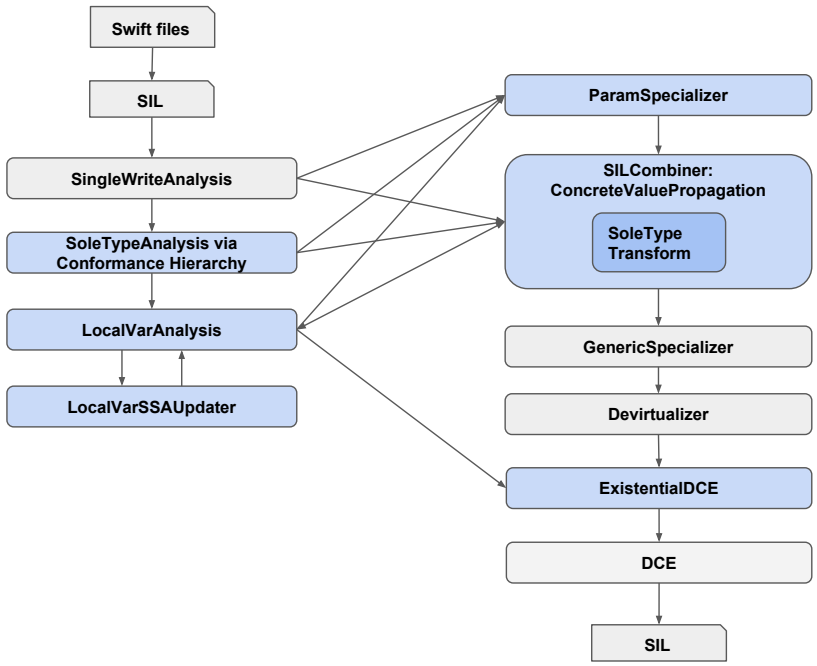

Fig. 4. Overall implementation framework of our existential optimizations. Colored boxes refer to passes that we have introduced in this paper except SILCombiner pass which we have extended.

parameter. Specifically, if at a call site we can determine a singleton concrete value or a sole concrete type for the protocol argument of a callee, we decide to apply `Param` to the callee. We also introduce the `ExistentialDCE` pass to remove unnecessary dead codes related to existentials as described in Section 4.5.

We extend the `SILCombiner` pass to consume the outputs of `SoleTypeAnalysis` and `LocalVarAnalysis` when rewriting call sites. The `SoleType` transformation determines optimal locations for inserting unchecked_ref_cast and uncheck_addr_cast instructions using the concrete value results from `LocalVarAnalysis` (as described in Section 4.4). `LocalVar` may be repeated based on the code modifications performed by `SoleType`. The baseline Swift compiler today performs a basic dominator-based single-write analysis for existentials (shown in Figure 4 as `SingleWriteAnalysis`) and use it to devirtualize or specialize certain existential operations. Finally, we leverage existing `GenericSpecializer` and `Devirtualizer` to specialize and devirtualize existential arguments.

6 EVALUATION

In this section, we present an experimental evaluation of the optimizations described in Section 4. Our evaluation aimed to evaluate the impact of the optimizations on both running time and code size. Here we detail our execution platform and methodology, and present results on both smaller benchmarks and a large production mobile app code base.

Platform: All results were obtained on an iMac Pro 2017 model equipped with 18-Core Intel Xeon W processor executing at 2.3GHz frequency with 64GB DDR4 memory, running Mac OS Mojave v10.14.4.

Benchmarks: We studied the impact of optimizing existential overheads on Swift Benchmark Suite [SwiftBench 2019] and BucketSort from [BucketSort 2019]. We only consider those applications from the Swift Benchmark Suite that use protocols and compile with the baseline compiler. We also evaluated impacts on a large production application, described in Section 6.3. The Swift Benchmark Suite benchmarks are used by the Swift developers to check for performance regressions in generated code. Thus, they provide evidence that the patterns seen in the benchmarks are common in real-world applications.

Methodology: The optimization framework described in Section 5 was implemented in the master branch of Swift compiler version 5.0-dev [SwiftLang 2019].[12] We report experimental results for the following cases:

(1) `Baseline` – the baseline version without any of the optimizations described in this paper;
(2) `Param` – the optimized version that uses the existential parameter specialization described in Section 4.3.
(3) `LocalVar` – the optimized version that uses the data-flow based optimization technique described in Section 4.2.
(4) `SoleType` – the optimized version that uses the sole-type based optimization technique described in Section 4.4.

We also report results for all combinations of the above options including optimization option "-O -wmo". We enable whole module optimization, i.e., "-wmo", for all our evaluations since it enables inter-procedural optimizations. While `Param` and `LocalVar` do not necessarily need `-wmo` mode, `SoleType` needs to analyze all the type declarations within a module. Additionally, we measure code size impact by measuring the size of the text segment using the command "size -m".

The execution times reported were the average of five separate runs for each benchmark[13]. Since all benchmarks studied in this paper are sequential, the results are not impacted by the number of cores.

Static metrics: Table 3 presents both source-level and compile-time characteristics of our benchmarks. Column 2 reports the number of protocols used in a benchmark. Column 3 reports LOC. Column 4 reports input size. Column 5 summarizes the impact of various optimizations statically. `BucketSort` is the largest benchmark in terms of LOC and uses 4 protocols. Six out of the 13 benchmarks have methods that are optimized using `Param`. Using `SoleType`, four call sites were devirtualized and 17 call sites were specialized. The `ObserverForwarderStruct` benchmark shows benefit from `LocalVar` by specializing a call site. Three benchmarks including `ObserverForwarderStruct`, `ProtocolDispatch2` and `SortLargeExistentials` show benefits from our dead code elimination using `LocalVar`. Please note that we only report copy_addr and store instructions removed by our `ExistentialDCE` pass, but in reality it may lead to the removal of other existential related dead code using the existing `DCE` pass. Overall, static metrics show each of our optimizations to be having some impact on some subset of the benchmarks.

6.1 Performance Results

Table 4 reports the relative speedup for each configuration compared to `Baseline` for each benchmark using the optimization flag "-O -wmo" on the iMac Pro 2017 system. The second column reports the absolute runtime for each benchmark in seconds for the `Baseline`. The remaining columns, i.e., columns 3-9, report speedup. `BucketSort` and `ProtocolDispatch2` benchmarks

[12] Specific versions of the base code are as follows: git sha 82c33dc0311a8874c333c8478d9c7251a21417ec, LLVM 6ddb64316c, Clang 8bf0fa1829, and Swift b5aabd1747

[13] We have observed negligible variations between 1-2% from run-to-run in our measurements.

Table 3. Static metrics using optimization flag "-O -wmo". Column 5 summarizes the impact of various optimizations statically for the optimizations described in this paper.

Benchmarks	#Protocols	LOC	Input Size	Optimization Summary
ArrayOfGenericRef	1	111	10000	No change
ArrayOfRef	1	122	10000	No change
Codable	2	168	10000	No change
DictionarySubscriptDefault	1	133	10000	No change
NSError	1	60	10000	No change
ObserverForwarderStruct	1	67	10000	1 method specialized via `Param`; 1 call site devirtualized by `SoleType`; 1 call site specialized by `LocalVar`; 1 store instruction dead
ObserverUnappliedMethod	1	69	10000	2 methods optimized via `Param`
PopFrontGeneric	1	82	10000	No change
ProtocolDispatch2	1	87	10000	2 methods specialized via `Param`; 12 call sites specialized via `SoleType`; 2 call sites devirtualized via `SoleType`; 2 load instructions dead;
SortLargeExistentials	1	106	1000	1 method specialized via `Param`; 4 call sites specialized via `SoleType`; 1 call site devirtualized via `SoleType`; 4 store instructions dead
StackPromo	1	74	10000	1 method specialized via `Param`; 1 call site specialized via `SoleType`
StaticArray	1	109	1M	No change
BucketSort	4	249	10M	2 methods specialized via `Param`

Table 4. Speedup of various combinations of optimizations compared to `Baseline` using optimization flag "-O -wmo". Column 2 reports the absolute runtime for each benchmark in seconds for the `Baseline` configuration. Columns 3-9 report speedups for various combination of optimizations vs. `Baseline`.

Benchmark	Baseline (sec)	Param	SoleType	LocalVar	Param+ SoleType	Param+ LocalVar	SoleType+ LocalVar	Param+ SoleType+ LocalVar
ArrayOfGenericRef	3.5	1.01	1.01	1.01	1.01	1.00	1.01	1.01
ArrayOfRef	3.42	1.00	1.00	1.00	1.00	1.00	1.00	1.01
Codable	10.6	1.00	1.01	1.00	1.01	1.01	1.00	1.01
DictionarySubscriptDefault	9.12	1.01	1.00	1.01	1.00	1.00	1.01	1.01
NSError	3.15	1.01	1.01	1.00	1.02	1.02	1.01	1.00
ObserverForwarderStruct	2.26	1.08	1.04	1.00	1.01	1.05	1.02	1.11
ObserverUnappliedMethod	2.3	1.12	1.02	1.01	1.11	1.11	1.01	1.12
PopFrontGeneric	0.4	1.00	1.00	1.00	1.00	1.00	0.99	1.00
ProtocolDispatch2	10.84	7.83	4.22	1.00	7.77	7.81	4.23	7.81
SortLargeExistentials	8.37	1.00	1.17	1.00	0.47	0.99	1.14	1.12
StackPromo	2.03	0.98	1.42	1.00	1.34	0.98	1.41	1.33
StaticArray	2.05	0.99	0.99	1.00	0.99	0.98	1.00	1.00
BucketSort	3.83	21.68	1.02	1.01	21.75	21.66	1.00	21.84
GEOMEAN		**1.50**	**1.17**	**1.00**	**1.45**	**1.50**	**1.16**	**1.56**

show significant performance benefits with `Param` configuration, i.e., 21.75× and 7.77× respectively. Both benchmarks exhibit method invocations with existential parameters in the hot loop, making them perfect candidates for `Param`. Overall, `Param` and `SoleType` show average speedups of 1.5× and 1.17×, respectively. Across the board for all optimization configuration combinations, we observe an average speedup of 1.56× when all the three optimizations are combined. In particular, `LocalVar` is able to eliminate a number of existential related deadcodes (static metrics are shown in the Table 3) resulting in an additional 6% performance improvement on top of `Param`.

```
156 func testStackAllocation(_ p: Protocol) -> Int {
157   var a = [p, p, p]
158   var b = 0
159   a.withUnsafeMutableBufferPointer {
160     let array = $0
161     for i in 0..<array.count {
162       b += array[i].at()
163     }
164   }
165   return b
166 }
```

Listing 15. Param Specializer Counter Example

Interestingly, while SortLargeExistential shows a maximum benefit when SoleType is applied, it demonstrates a degradation of 5% when all the three optimizations are combined together. We found that this was due to the phase ordering problem between optimizations, i.e, SoleType transforms the generic methods produced by Param before these methods are specialized via GenericSpecializer, leading to a sub-optimal code sequence. Ideally, one would like to specialize the Param-generated methods before applying SoleType. This behavior is also clearly evident when Param+SoleType observes a speedup of 0.47× in SortLargeExistential. However, this slowdown disappears when LocalVar optimization is enabled for this benchmark. Fixing the phase ordering issue requires a significant restructuring of the Swift compiler passes and is left as a future work.

While a similar behavior is also observed in StackPromo, this benchmark also had a slowdown because of the boxing operations introduced by Param, which were not eliminated since the existential was stored into an array. The code snippet for this is shown in Listing 15. In this example, the protocol parameter p is stored into an array in Line 112 and subsequently accessed out of the array in Line 117. Even though p can be specialized to a concrete type, it is clearly beneficial to keep p in existential form. Param converts p into a generic parameter and later on boxes it to build an existential (like Listing 9), which leads to additional boxing overhead and thus the slowdown. In future, we would like to enhance the profitability of Param to handle this scenario.

6.2 Code Size

Table 5 presents code size improvements in bytes for various configurations. The last column reports the size reduction for the configuration with all optimizations applied together compared to Baseline. All benchmarks that observe performance benefits observe a reduction in code size except BucketSort. We found out that BucketSort defines the method bucketSort as **public** (i.e., another Swift module can access this method) which means although this method gets specialized by Param, its original version still remains in the final binary along with the specialized version. Thus we observe an increase in code size even though we significantly improve the performance for this benchmark. If we change the access level of this method from **public** to **internal**, we observe a code size decrease of 758 bytes[14]. Overall, the data show that on these benchmarks, the performance improvements from our optimizations do not come at a cost of code bloat.

6.3 Proprietary Large Swift application

We also evaluate the benefits of our optimization on a large proprietary iOS application from company Uber written mostly using Swift. The total LOC for Uber's Rider application is 1.7MLoC, out of which Swift code constitutes 83% approximately and the remaining code is written using

[14]Only Param optimization suffers from this problem, but not SoleType or LocalVar.

Table 5. Code size savings in bytes using optimization flag "-O -wmo". Columns 2-9 report actual bytes used for different optimization combinations. Column 10 reports the absolute code size savings in bytes.

Benchmark	Baseline	Param	SoleType	LocalVar	Param+ SoleType	Param+ LocalVar	SoleType+ LocalVar	Param+ SoleType+ LocalVar	Diff Baseline - (Param+ SoleType+ LocalVar)
ArrayOfGenericRef	14251	14251	14251	14251	14251	14251	14251	14251	0
ArrayOfRef	11035	11035	11035	11035	11035	11035	11035	11035	0
Codable	22010	22010	22010	22010	22010	22010	22010	22010	0
DictionarySubscriptDefault	26155	26155	26155	26155	26155	26155	26155	26155	0
NSError	2715	2715	2715	2715	2715	2715	2715	2715	0
ObserverForwarderStruct	4720	4656	4527	4720	4607	4656	4463	4415	**305**
ObserverUnappliedMethod	5865	5817	5865	5865	5817	5817	5865	5817	**48**
PopFrontGeneric	6085	6085	6085	6085	6085	6085	6085	6085	0
ProtocolDispatch2	4336	3645	3823	4336	3645	3645	3823	3645	**691**
SortLargeExistentials	19832	19832	19768	19832	20024	19832	19464	19512	**320**
StackPromo	3456	3468	3285	3456	3269	3468	3285	3269	**187**
StaticArray	14518	14518	14518	14518	14518	14518	14518	14518	0
BucketSort	12866	14114	12866	12866	14114	14114	12866	14114	**-1248**

Objective-C. Our optimizations only target Swift code. The application is used by millions of customers every day. Although it is not possible to release the source code of this proprietary application, prior versions of the core architecture of this application has been released in the open source [UberRibs 2017]. Nevertheless, we also present results from a micro-benchmark that mimics some of the functionalities of the large app.

Since the application does not yet compile on the latest Swift compiler version 5.0, we backported our optimizations to Swift version 4.2. We report execution time results on a iPhone 6S device. We compare the following approaches:

(1) `UNOPT` using optimization flag "-Osize -wmo";
(2) `OPT` using optimization flag "-Osize -wmo" and `Param+SoleType+LocalVar`.

Static Metrics: The core components of the application use a large number of protocols. In total, they use 9991 swift protocols out of which 4992 are declared as **`public`** and the rest are either **`internal`** or **`private`**. Many of the protocols are used for test mocking, i.e., a protocol has two implementations, one for testing in debug mode and another for production mode. Our `OPT` configuration specialized 6296 methods using `Param`. Furthermore, it devirtualizes a total of 23418 call sites and specializes arguments of 88 call sites using `SoleType` statically. The `LocalVar` helps in removing 1066 number of `copy_addr` and `store` instructions in addition to other existential operations that are proved dead using existing `DeadCodeElimination` pass.

Runtime improvements: We ran cold startup of the app as well as the most frequently used core-functionality of the application 15 times on the iPhone 6S device and report average timings for different spans using `UNOPT` and `OPT` configurations as shown in Figure 5. These performance spans were inserted by the app developers[15] and they are used to track performance regressions in the field. We only report timings for spans that take more than 100 milliseconds.

Overall, we observe significant positive speedups for all spans in the core functionality, with a minimum improvement of 6.88% to a maximum improvement of 55.49%. For the `cold-startup`[16], we observe an improvement of 12%. We notice an improvement of 13% for the premain time (this is the time is spent before the app code gets executed, e.g., loading dylibs, rebasing, and binding) which is primarily due to the reduction in protocol witness tables from the final binary after our optimizations are applied.

[15] None of the authors were involved in defining the performance spans.
[16] *cold-startup* refers to launching the app for the first time and that there is no caching for the app in the kernel. This is the worst case behavior for the app launch.

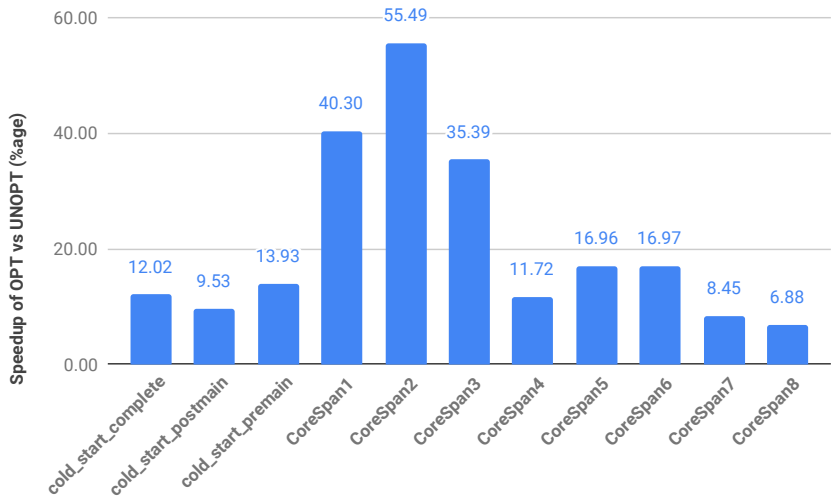

Fig. 5. Runtime benefits (in percentage) for cold startup and core functionalities using optimization flag "-Osize -wmo" for the large proprietary Swift app. The same sequence of operations were performed 15 times for both UNOPT and OPT on the iPhone 6S mobile device.

Codesize reduction: Although specialization techniques presented in this paper could potentially lead to code bloat, we actually observed a code size *reduction* of 1.7% and 1.4% on armv7 and arm64 architectures, respectively. This result is very encouraging, as excessive code size bloat is unacceptable for a production mobile app at scale.

Micro-benchmark: We are unable make Uber's Rider app code base public. Instead, we have constructed a micro-benchmark[17] that mimics some of the common protocol usage scenarios of the real app. On a 2018 Macbook Pro system comprising of 2.6 GHz Intel Core i7 and 32GB RAM running Mac OS 10.14.6 and a Swift compiler v5.1-dev[18], we observe a speedup of 11.76% with all the three optimizations described in this paper (i.e., `LocalVar` + `Param` + `SoleType`) compared to the Baseline ("-O -wmo").

7 RELATED WORK

In this section, we compare our Swift protocol optimizations with prior work on eliminating virtual call overheads in different settings.

Devirtualizing method calls is one of the important optimizations in dynamically typed languages [Ahn et al. 2014; Anderson et al. 2011; Deutsch and Schiffman 1984; Dot et al. 2017; Gal et al. 2009; Hölzle et al. 1991; Hölzle and Ungar 1994] and most of the techniques perform some form of "inline caching". These techniques typically rely on observing call targets inside a JIT compiler. Swift uses ahead-of-time compilation, but we plan to investigate integrating profile-guided optimization to obtain further benefits. Note that inline caching aids in devirtualization, but it is insufficient to eliminate the overheads of boxing and unboxing from existential containers.

[17] https://github.com/rajbarik/OOPSLA-2019-SampleApp

[18] Based on git commit sha 44af3a9398914ef8bdb92eea37d235a92be60925 from [SwiftLang 2019]; please note that we also had to port our changes to Swift 5.1 branch for this evaluation.

It is well known that for statically typed languages, method call overhead is negligible [Ishizaki et al. 2000], but the ability to inline the callee and specialize it based on the concrete type information yields higher benefits. Although Swift is statically typed, the protocol support for handing both reference types and value types makes their sizes unknown statically and hence involves runtime resolution via boxing and unboxing.

A large body of work [Bacon and Sweeney 1996; Dean et al. 1995; Fernández 1995; Sundaresan et al. 2000] performs *class-hierarchy analysis* to identify concrete types to optimize virtual method calls. In particular, Dean et al. [Dean et al. 1995] perform a whole-program analysis to establish the complete class hierarchy. They then use this information to replace virtual method calls with static calls at call sites where the receiver type can be proven to be in the class hierarchy sub-tree that has no overrides for the the virtual function under question. They apply their technique to both statically and dynamically typed languages. This technique requires no data-flow analysis and hence is fast and avails itself to incremental compilation. The `SoleType` technique described in this paper leverages a similar type-hierarchy analysis, but since our analysis operates at a module granularity instead of on the whole program, it only analyzes protocols internal to the module. More importantly, our `SoleType` pass determines optimal places to inject the casts based on `LocalVar`.

A large body of work [Agesen et al. 1993; Bacon and Sweeney 1996; Hirzel et al. 2007; Jagannathan and Weeks 1995; Shivers 1988, 1991; Steensgaard 1996] performs inter-procedural points-to analysis to determine the set of classes that might be stored into a variable and use that information to improve the accuracy of devirtualization. In the future, we could leverage similar techniques along with the `SoleType` transformation to achieve further performance gains. Our `LocalVar` technique is related to this body of work, but it operates purely intra-procedurally, and it focuses only on concrete values stored and manipulated in existential containers.

Java programs have received significant attraction in method devirtualization and focused on fast techniques [Cierniak et al. 2000; Detlefs and Agesen 1999; Ishizaki et al. 2000] in the presence of partial class hierarchy information. These techniques, typically, devirtualize and inline based on prediction or a previous execution history and allow "back patching" when the speculation fails. Our techniques are orthogonal to these techniques; we do not employ speculation and back patching. Moreover, code patching at runtime is not an option for compiled binaries in the iOS mobile app distribution. Similar to the previously discussed techniques, the optimizations in Java do not circumvent the boxing and unboxing concerns that exist in Swift.

Specialization is another common optimization in Java and other object-oriented languages [Fujinami 1998; Kedlaya et al. 2013; Masuhara and Yonezawa 2002; Rompf et al. 2014; Schultz et al. 2003]. Our parameter optimization (`Param`) identifies the places for optimization based on concrete types at call sites and relies on Swift's generic specialization to generate the specialized code variants.

Type classes in Haskell [Hudak and Fasel 1992, §5] are the generic interfaces that provide a common feature set over many types. Haskell performs "monomorphization", which transforms a type-checked Haskell program with type classes into the code without type classes or bounded polymorphism. Monomorphization recursively substitutes all overloaded identifiers with whatever they resolve to until no overloading is left. Monomorphization eliminates the run-time overhead associated with dynamic dispatch, however, monomorphization requires whole program analysis and does not work with separate compilation units. Our repeated application of the `Param` optimization bears resemblance with monomorphization, but we neither guarantee 100% substitution nor require whole program analysis.

Traits in the Rust programming language [Klabnik and Nichols 2019, §10.2] are similar to protocols in Swift, in that both value and reference types may implement the same trait. However, by default Rust does *not* perform boxing and unboxing corresponding to existential containers to generate code that can handle arbitrary concrete types implementing a trait. Instead, programmers

must opt in to such boxing explicitly, using trait objects [Klabnik and Nichols 2019, §17.2]. So, the overhead concerns this paper addresses for Swift should be less prevalent in Rust.

8 CONCLUSION

In this paper, we have highlighted performance inefficiencies associated with protocols in the Swift language. Due to the versatility of protocols, they can introduce both dynamic dispatch and boxing overheads on protocol-heavy code. We have described three key optimizations for analyzing and optimizing protocol-related code at the intermediate-language level. Overall, our optimizations improve runtime performance by 1.56× on average for a set of 13 benchmarks obtained from Swift Benchmark Suite. Moreover, we demonstrate a speedup ranging from 6.9% to 55.49% for performance spans in a production mobile application consisting of close to 1.5MLOC Swift code.

ACKNOWLEDGMENTS

We thank Slava Pestov, Arnold Schwaighofer and Andrew Trick from Apple for their help and support in incorporating our optimizations into the Swift Compiler. We are also thankful to the anonymous reviewers for their insightful comments and suggestions. We are grateful to Ellie Shin, Rudro Samanta, Mahyar McDonald, and Richard Howell from Uber for their engagement in both uncovering the Swift protocol overhead problem in the Rider mobile app and performing experiments on real devices.

REFERENCES

Ole Agesen, Jens Palsberg, and Michael I. Schwartzbach. 1993. Type Inference of Self. In *ECOOP' 93 — Object-Oriented Programming*, Oscar M. Nierstrasz (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 247–267. https://doi.org/10.1007/3-540-47910-4_14

Wonsun Ahn, Jiho Choi, Thomas Shull, María J. Garzarán, and Josep Torrellas. 2014. Improving JavaScript Performance by Deconstructing the Type System. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, New York, NY, USA, 496–507. https://doi.org/10.1145/2594291.2594332

Owen Anderson, Emily Fortuna, Luis Ceze, and Susan Eggers. 2011. Checked load: Architectural support for javascript type-checking on mobile processors. In *2011 IEEE 17th International Symposium on High Performance Computer Architecture*. IEEE, 419–430. https://doi.org/10.1109/HPCA.2011.5749748

AppleOTA 2019. Apple Over-The-Air Requirements. https://developer.apple.com/accessories/Accessory-Design-Guidelines.pdf. Accessed: 2019-04-01.

David F. Bacon and Peter F. Sweeney. 1996. Fast Static Analysis of C++ Virtual Function Calls. In *Proceedings of the 11th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '96)*. ACM, New York, NY, USA, 324–341. https://doi.org/10.1145/236337.236371

BucketSort 2019. BucketSort. https://github.com/raywenderlich/swift-algorithm-club/tree/master/Bucket%20Sort. Accessed: 2019-01-29.

MichaÅĆ Cierniak, Guei-Yuan Lueh, and James M. Stichnoth. 2000. Practicing JUDO: Java Under Dynamic Optimizations. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI '00)*. ACM, New York, NY, USA, 13–26. https://doi.org/10.1145/349299.349306

Jeffrey Dean, David Grove, and Craig Chambers. 1995. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In *ECOOP'95 — Object-Oriented Programming, 9th European Conference, Åarhus, Denmark, August 7–11, 1995*, Mario Tokoro and Remo Pareschi (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 77–101. https://doi.org/10.1007/3-540-49538-X_5

David Detlefs and Ole Agesen. 1999. Inlining of Virtual Methods. In *Proceedings of the 13th European Conference on Object-Oriented Programming (ECOOP '99)*. Springer-Verlag, London, UK, UK, 258–278. https://doi.org/10.1007/3-540-48743-3_12

L. Peter Deutsch and Allan M. Schiffman. 1984. Efficient Implementation of the Smalltalk-80 System. In *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '84)*. ACM, New York, NY, USA, 297–302. https://doi.org/10.1145/800017.800542

Gem Dot, Alejandro Martínez, and Antonio González. 2017. Removing Checks in Dynamically Typed Languages Through Efficient Profiling. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization (CGO '17)*. IEEE Press, Piscataway, NJ, USA, 257–268. https://doi.org/10.1109/CGO.2017.7863745

Mary F. Fernández. 1995. Simple and Effective Link-time Optimization of Modula-3 Programs. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation (PLDI '95)*. ACM, New York, NY, USA, 103–115. https://doi.org/10.1145/207110.207121

Nobuhisa Fujinami. 1998. Determination of Dynamic Method Dispatches Using Run-Time Code Generation. In *Proceedings of the Second International Workshop on Types in Compilation (TIC '98)*. Springer-Verlag, Berlin, Heidelberg, 253–271. https://doi.org/10.1007/BFb0055522

Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R. Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, Jesse Ruderman, Edwin W. Smith, Rick Reitmaier, Michael Bebenita, Mason Chang, and Michael Franz. 2009. Trace-based Just-in-time Type Specialization for Dynamic Languages. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*. ACM, New York, NY, USA, 465–478. https://doi.org/10.1145/1542476.1542528

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Martin Hirzel, Daniel Von Dincklage, Amer Diwan, and Michael Hind. 2007. Fast Online Pointer Analysis. *ACM Trans. Program. Lang. Syst.* 29, 2, Article 11 (April 2007). https://doi.org/10.1145/1216374.1216379

Jon Hoffman. 2019. *Swift 4 Protocol-Oriented Programming* (3rd ed.). Packt Publishing.

Urs Hölzle, Craig Chambers, and David Ungar. 1991. Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP '91)*. Springer-Verlag, London, UK, UK, 21–38. https://doi.org/10.1007/BFb0057013

Urs Hölzle and David Ungar. 1994. Optimizing Dynamically-dispatched Calls with Run-time Type Feedback. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation (PLDI '94)*. ACM, New York, NY, USA, 326–336. https://doi.org/10.1145/178243.178478

Paul Hudak and Joseph H. Fasel. 1992. A Gentle Introduction to Haskell. *SIGPLAN Not.* 27, 5 (May 1992), 1–52. https://doi.org/10.1145/130697.130698

Kazuaki Ishizaki, Motohiro Kawahito, Toshiaki Yasue, Hideaki Komatsu, and Toshio Nakatani. 2000. A Study of Devirtualization Techniques for a Java Just-In-Time Compiler. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '00)*. ACM, New York, NY, USA, 294–310. https://doi.org/10.1145/353171.353191

Suresh Jagannathan and Stephen Weeks. 1995. A Unified Treatment of Flow Analysis in Higher-order Languages. In *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '95)*. ACM, New York, NY, USA, 393–407. https://doi.org/10.1145/199448.199536

Madhukar N. Kedlaya, Jared Roesch, Behnam Robatmili, Mehrdad Reshadi, and Ben Hardekopf. 2013. Improved Type Specialization for Dynamic Scripting Languages. In *Proceedings of the 9th Symposium on Dynamic Languages (DLS '13)*. ACM, New York, NY, USA, 37–48. https://doi.org/10.1145/2508168.2508177

Steve Klabnik and Carol Nichols. 2019. The Rust Programming Language. https://doc.rust-lang.org/stable/book/. Accessed: 2019-04-05.

William Landi and Barbara G. Ryder. 1991. Pointer-Induced Aliasing: A Problem Classification. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages, Orlando, Florida, USA, January 21-23, 1991*. 93–103. https://doi.org/10.1145/99583.99599

Hidehiko Masuhara and Akinori Yonezawa. 2002. A Portable-approach to Dynamic Optimization in Run-time Specialization. *New Gen. Comput.* 20, 1 (Jan. 2002), 101–124. https://doi.org/10.1007/BF03037261

Benjamin C. Pierce. 2002. *Types and Programming Languages* (1st ed.). The MIT Press.

Tiark Rompf, Arvind K. Sujeeth, Kevin J. Brown, HyoukJoong Lee, Hassan Chafi, and Kunle Olukotun. 2014. Surgical Precision JIT Compilers. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, New York, NY, USA, 41–52. https://doi.org/10.1145/2594291.2594316

Ulrik P. Schultz, Julia L. Lawall, and Charles Consel. 2003. Automatic Program Specialization for Java. *ACM Trans. Program. Lang. Syst.* 25, 4 (July 2003), 452–499. https://doi.org/10.1145/778559.778561

O. Shivers. 1988. Control Flow Analysis in Scheme. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation (PLDI '88)*. ACM, New York, NY, USA, 164–174. https://doi.org/10.1145/53990.54007

Olin Grigsby Shivers. 1991. *Control-flow Analysis of Higher-order Languages of Taming Lambda*. Ph.D. Dissertation. Pittsburgh, PA, USA. UMI Order No. GAX91-26964.

SILDocs 2019. Swift Intermediate Language (SIL). https://github.com/apple/swift/blob/master/docs/SIL.rst. Accessed: 2019-03-22.

SILSSA 2017. SIL SSA Updater. https://github.com/apple/swift/blob/master/lib/SILOptimizer/Utils/SILSSAUpdater.cpp. Accessed: 2019-01-29.

SILVerifier 2017. SILVerifier. https://github.com/apple/swift/blob/master/lib/SIL/SILVerifier.cpp. Accessed: 2019-01-29.

Bjarne Steensgaard. 1996. Points-to Analysis in Almost Linear Time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '96)*. ACM, New York, NY, USA, 32–41. https://doi.org/10.1145/237721.237727

Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. 2000. Practical Virtual Method Call Resolution for Java. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '00)*. ACM, New York, NY, USA, 264–280. https://doi.org/10.1145/353171.353189

SwiftBench 2019. Swift Benchmark Suite. https://github.com/apple/swift/tree/master/benchmark. Accessed: 2019-01-29.

SwiftLang 2019. Swift Programming Language. https://github.com/apple/swift/. Accessed: 2019-01-29.

SwiftLangDoc 2019. Swift Language Documentation. https://developer.apple.com/documentation/swift Accessed: 2019-04-01.

UberRibs 2017. Ribs: Cross Platform Mobile Architecture. https://github.com/uber/RIBs/. Accessed: 2019-01-29.

WWDCSwiftPerf 2016. Understanding Swift Performance - Apple WWDC 2016. https://developer.apple.com/videos/play/wwdc2016/416/. Accessed: 2019-01-29.