Partial Specialization for Subtype Polymorphism in Virgil

Yonah Goldberg (ygoldber) **Jason Yao (jlyao)**

Project URL: https://yonahg.com/optcomp

Project PR: https://github.com/titzer/virgil/pull/309

I. Introduction

Object oriented programs often rely on subtyping to enable code reuse, abstraction, and modularity. Subtyping enables polymorphism, allowing a subclass to be treated as if it were an instance of its superclass. You can write general-purpose code that operates on a parent type, and the specific behavior is determined at runtime based on the actual subtype.

Consider the following example program:

```
class Animal {
  def speak() -> string;
}
class Pig extends Animal {
  def speak() -> string { return "oink"; }
}
class Cow extends Animal {
  def speak() -> string { return "moo"; }
}
def accept(A: Animal) -> string {
  return A.speak();
}
```

Here, `A.speak()` is a virtual function call. The actual behavior of the call depends on the type of animal passed as argument to `accept`. The function that runs is selected via a virtual dispatch. The compiler creates a virtual table for the `speak()` method. At runtime, the program dereferences `A` to lookup its class ID (ie. 0 if it is a `Pig` and 1 if it is a `Cow`). It then indexes into the vtable by class ID to obtain the correct method implementation for the type.

In general, virtual dispatches require two extra pointer dereferences, which can be costly. Virtual method calls also impose a barrier to inlining, an important optimization.

I.A. Specialization

Virtual method calls can sometimes be eliminated by method specialization. Specialization is the process of generating optimized versions of code tailored for specific use cases or contexts. In this case, we perform subtype specialization. We can create two new versions of `accept`:

```
def acceptPig(P: Pig) -> string {
  return P.speak();
}
def acceptCow(C: Cow) -> string {
  return C.speak();
}
```

If the compiler can determine the type of animal at the `accept` call-site, it can replace the call with a call to the optimized accept method. If it cannot determine the type of animal, it may still be beneficial to case on the type and then call the more specialized method, if the specialized method is sufficiently more performant.

I.B. Approach

We implemented subtype specialization in `Virgil`, a statically typed, object oriented, research language designed by Carnegie Mellon professor Ben Titzer. The previous code snippets are written in `Virgil`.

One of the benefits of `Virgil` is it is whole program compiled and optimized. Most languages compile code in units, separated by source files or modules, and later link them together. The benefit of whole program compilation is that the compiler knows the entire class hierarchy and all call sites of every method. This way, it has complete information to determine how to specialize.

Our first implementation is inefficient and specializes all methods to all parameter subtypes for all parameters. Our second implementation is smarter, and only specializes methods to subtype combinations we know they are called with.

I.C. Related Work

Specialization is commonly applied to JIT compiled lanaguges. The `Julia` programming language is JIT compiled and agressively specializes functions after they are called [1].

One of the earliest works on specialization was in the SELF language, which is dynamically typed and object oriented. [2].

Our version is more restricted since we only specialize instances of subtype polymorphism and specialization is performed at compile-time. It is unique because `Virgil` is whole program compiled, so we can benchmark large programs where there are potentially many opportunities to specialize.

I.D. Contribution

Our project contributes an implementation of a new subtype specialization optimization for the `Virgil` compiler and provides the infrastructure to extend the optimization with more heuristics and specialization opportunities. As a part of this optimization, we compute the call-graph of a program, which can be used for additional interprocedural optimizations.

II. The `Virgil` Programming Language

II.A. Language Features

As mentioned before, `Virgil` is primarily a statically-typed, object-oriented language. However, it does also support algebraic datatypes in addition to classes, as well as pattern matching on both datatypes and classes.

Support for lambdas/closures is still in progress, but functions and class methods are considered values and partial application is possible to support some sense of higher-order functions.

```
def foo(x: int, y: int) { return x + y; }
def f: int -> int = foo(_, 1);
```

In this example, we partially apply `foo` with `1` as its second argument and assign `f` to the new function value. `f` can be later applied like `f(2)`, which would then evaluate `2 + 1` and return it.

For this paper, we will generally use the term "method" to refer to both functions and methods since internally, top-level functions are represented as methods with no receivers.

`Virgil` also has support for type parameters, similar to generics in Java.

II.B. Compiler Design

The `Virgil` compiler is entirely self-hosted and targets x86 and x86-64 on both Linux and Darwin kernels, as well as JVM and WASM. Since it is a whole-program optimizing compiler, there is no linking stage; the binary is directly generated by the compiler.

After parsing and typechecking, the compiler lowers the source code to an SSA IR where it performs the majority of analysis and optimizations. One major analysis is "reachability", where a simple context and flow-insensitive analysis is used to determine all possible live classes and methods. Here, all the classes are assigned an unique index, which is used to generate and index into virtual tables for virtual method dispatches. This is only possible since the entire program is compiled at once.

After reachability, a "normalization" phase occurs, where type arguments are substituted in to greatly reduce code complexity. We perform our optimizations after this phase since we rely on reachability analysis to determine all live subclasses of each class. We also wished to specialize methods after type arguments were substituted in so we had the most specific type information within method bodies.

After normalization, the IR is lowered to Machine SSA, a relatively generic IR that takes into account machine-specific information such as pointer size and eventually transformed in to code for the target architecture.

III. Specialization Algorithm

We begin by discussing how to specialize a method. The first step is to duplicate the method's signature, replacing polymorphic types with the more specific variants we are specializing to. Then you walk the method's body. In most cases, we can simply copy instructions to form the new specialized method's body. If we see a virtual method call or a virtual field access, we replace it with the method implementation or field of the type we are specializing to. We also importantly replace the type argument of type queries.

Consider the following contrived example:

```
def accept(A: Animal) -> string {
  match (A) {
    x: Pig => return "a pig";
    x: Cow => return x.speak();
  }
}
```

`Virgil` supports pattern matching, and methods sometimes match on the type of a parameter. Consider the case that we specialize `accept` to the `Cow` subtype. The match clause resolves to a sequence of type queries for which we replace the type argument. The virtual call to `x.speak()` is replaced with a direct call to the `Cow` implementation.

We then run the `Virgil` inliner, which will inline the `x.speak()` call.

We then run the `Virgil` SSA optimizer, which will see type queries to known types and it folds away the match clause. The optimizer also performs constant/copy propagation, constant folding, strength reduction, and other optimizations not used here, but are generally useful after inlining. In the end, we output the following specialized version of the method:

```
def acceptCow(C: Cow) -> string {
  return "moo";
}
```

III.A. Initial Design

Our initial design specialized all methods to all potential subtypes with all possible combinations. For example, the following method would receive 4 specializations:

```
def acceptTwo(a1: Animal, a2: Animal) { ... }
# specializes to..
def acceptTwoPigPig(a1: Pig, a2: Pig) { ... }
def acceptTwoPigCow(a1: Pig, a2: Cow) { ... }
def acceptTwoCowPig(a1: Cow, a2: Pig) { ... }
def acceptTwoCowCow(a1: Cow, a2: Cow) { ... }
```

Using this implementation, we could debug our specialization code and observe a performance increase after running additional optimizations on the specialized methods.

Then, we needed to limit our specializations to a subset of the available subtypes in order to reduce code-size blowup and get rid of unnecessary specializations.

III.B. Deciding When To Specialize

We decided to only specialize methods that are called with arguments that are statically known to have a more specific subtype than its parameters. The `Virgil` type system can sometimes determine the exact subtype of an object with a class hierarchy analysis. There are other options to increase the power of specialization, such as inserting type queries to determine the exact type, followed by calling into the specialized method. More details about this can be found in Section VII.B.

To perform this inter-procedural analysis, we generate an undirected call-graph by traversing the program using depth-first search (DFS). The call-graph is represented using two adjacency lists, one for the `callerToCallee` direction and one for the `calleeToCaller` direction.

Edges in the call-graph correspond to individual call sites. For example, the following two methods:

```
1: def bar(i: int) -> int {
2:   return i * 2;
3: }
4: def foo(x: int) -> int {
5:   def a = bar(x + 1);
6:   def b = bar(3);
7:   return a - b;
8: }
```

would generate the following call-graph, where we reference the actual SSA instruction instead of a line number:

```
callerToCallee:
  foo -> bar (line 5), bar (line 6)

calleeToCaller:
  bar -> foo (line 5), foo (line 6)
```

III.C. Using the Call-Graph

Now that we have a call-graph of all the methods in the program, we can proceed with a worklist-style algorithm to iterate over methods and specialize them. Once we specialize a method's parameters, any use of that parameter in other calls could enable more specializations that were not possible before. For example, consider these two methods:

```
def accept2(A: Animal) -> string {
  match (A) {
    x: Pig => return "a pig";
    x: Cow => return x.speak();
  }
}

def accept(A: Animal) -> string {
  return accept2(A);
}
```

If we only make a single pass to specialize methods, we would fail to specialize `accept2` since its only call site is with the parent class `Animal`. However, if we first specialize `accept` to `Pig` or `Cow`, now we know `accept2` can be specialized to that same subtype. Thus, we have the worklist-style algorithm to add the call-targets of specialized methods to the worklist.

Initially, we add all methods that are targets of calls to the worklist. We then traverse each instruction that calls them and determine if any of the arguments are subtypes of the method parameters. If they are, that specific specialization is added to a set to ensure we do not specialize a method the same way twice.

Next, we generate all the specialized versions of the method from the set of desired specializations. We then traverse the call-sites again to rewrite any call instructions to the specialized method. If we rewrite the call instruction, we add a new edge to the call-graph, kill the old one, and traverse the specialized method to add all the methods it calls to the the call-graph. The callees of the specialized method are added back to the worklist to check if more specializations has been enabled for them.

IV. Evaluation

IV.A. Experiment Setup

We benchmark six `Virgil` programs, described in section Section IV.C and included in our PR in the `virgil/bench` directory. Each program was benchmarked on one of the Gates cluster machines. We test code-size and run-time of these programs in the following modes:

- **Not Specialized** is the `Virgil` compiler run on `O1`.
- **Call-Graph Specialized No Opt** runs specialization to eliminate virtual dispatches, but doesn't run the inliner or optimizer after the fact.
- **Call-Graph Specialized** runs specialization, followed by inlining and the SSA Optimizer.
- **Everything Specialized** specializes all methods to all potential subtype combinations.

IV.B. Running a Benchmark

For an example on how to replicate our results, you can run the following command in the `Virgil/bench` directory:

```
V3C_OPTS=-opt=SubtypeSpecialization,
SubtypeSpecializationNoOpt ./run.bash ../bin/
dev/v3c-dev large x86-64-linux TypeBench
```

`TypeBench` is the benchmark to run, `x86-64-linux` is the target, and `./bin/dev/v3c-dev` is the dev compiler. The `V3C_OPTS` environment variable allows you to set compiler options. We added three new compiler options:

1. `-opt=SubtypeSpecialization` runs full call-graph style specialization.
2. `-opt=SubtypeSpecialization, SubtypeSpecializationNoOpt` runs call-graph style specialization, but turns off inlining and optimizing after the fact.
3. `-print-subtype-specialization` prints the SSA of methods after they are specialized. You can combine this with `-print-ssa` to compare the before and after.

IV.C. The Benchmarks

Below are short descriptions of the six `Virgil` programs we benchmarked. Note that there are inherent limitations in trying to benchmark a research language where there are not many large applications written in it. Therefore, we use some contrived cases we wrote to showcase performance under ideal circumstances. The exact source of these benchmarks can be found in the GitHub pull-request.

IV.C.1. V3C

The `Virgil` compiler, which is probably the largest `Virgil` program that exists.

IV.C.2. TypeBench

Benchmarks constructing types that are represented similar to how they are in `V3C`. Includes a `HashMap` implementation for caching types, a lot of polymorhpism, and operations on collections.

IV.C.3. DeltaBlue

A constraint solver, based on the Java version by Mario Wolczko. Includes a lot of polymorphism, dynamic dispatch, and memory allocation and management.

IV.C.4. Contrived1

The first contrived test case. Includes a large inheritance hierarchy, and runtime dominated by a loop that calls a function that can be specialized. The specialized function calls a virtual function, that itself can be specialized after its caller is specialized.

IV.C.5. Contrived2

Includes a runtime dominated by a loop that calls a function that can be specialized. The specialized function has a large match clause that can be folded away after specialization.

IV.C.6. Contrived3

Similar to Contrived1, but one argument to the specialized function has a type that can be partially constrained at compile-time, but not completely, so that the specialized function still has to make a virtual dispatch.

IV.D. Experiment Results

IV.D.1. Runtime

Figure 1 shows performance of subtype specialization across our benchmarks. Notably, we struggle on the larger, non-contrived benchmarks, but achieve 2-3x speedup on the contrived benchmarks.

IV.D.2. V3C, TypeBench, And DeltaBlue Runtime

The runtime of V3C is dominated by large walks over the entire program, allocating and deallocating data. It is hard to show speedup with a niche optimization on such a large benchmark. TypeBench and DeltaBlue suffer similarly.

In analyzing the types of specializations the compiler performs on these programs, we see that there are many specialization opportunities. The compiler finds 43 in V3C and 2 each in TypeBench and DeltaBlue. The surprise was that only eliminating virtual method calls does not have a dramatic affect on performance. We see that in the contrived benchmarks, we only see large performance gains after further inlining and optimization. Modern hardware can take advantage of instruction level parallelism (ILP) and caching, hiding most of the latency of the virtual dispatch.

Unfortunately, V3C, TypeBench, and DeltaBlue do not contain many opportunites to inline and further optimize after specialization, which is why we do not see performance gains. If we had time, we would have tried to find a real world program that exibits patterns specialization can take advantage of. The problem with using a research language is that this takes extra time because we would have to port the program to `Virgil`.

IV.D.3. Contrived Benchmark Runtime

Our contrived benchmarks perform extremely well, obtaining up to 3x speedup. This is because their runtime is dominated by a loop that repeatedly calls a function that envokes many virtual dispatches. Specialization alone eliminates these virtual dispatches, causing some performance increase. As seen in Figure 1, the main performance increase comes from inlining and folding after specialization.

IV.D.4. Code Size

Figure 2 compares the differences in the size of the compiled binaries for the benchmark programs. We notice that in general, we do not dramatically increase the code size, even when specializing for all possible subtypes. For the more reasonable call-graph specialized case, the greatest increase in size comes from DeltaBlue, with about an 11% larger binary, while specializing everything increased the code size by almost 26%. These relatively small increases in code size partly influenced our decision to not pursue further heuristics that would further limit the amount of specialization beyond just the call-graph specialized case.

Figure 3 depicts the code size differences when compiling the compiler itself. We see a similar story for the call-graph case, however, specializing everything actually failed due to running out of heap memory. This makes sense, since the compiler heavily uses subtype polymorphism to represent each IR, so specializing everything generates almost an exponential increase in number of functions depending on their arguments.

Overall, we were slightly surprised about the relatively small code size impact our optimization had. This is likely due to the relatively limited cases where functions are called with more specific types that can be known statically at compile-time. All the concrete numbers for code size can be found in Table 2.

V. Surprises and Lessons Learned

V.A. Virtual Calls Alone Have Low Overhead

One of the first surprises we encountered was after we implemented the specialization code, but before we performed any further optimizations, like inlining, on the specialized code. We observed that virtual calls alone have low overhead, as mentioned in Section IV.D.2, and eliminating them did not have a dramatic affect on performance. This seemed counterintuitive since, in our contrived benchmark, we were eliminating millions of memory accesses for virtual table lookups. We suspect modern hardware is playing tricks like ILP and caching, minimizing the influence of the memory accesses.

V.B. Working in a Research Language is Hard

Working in `Virgil` has reinforced our appreciation for the existence of well-supported ecosystems for programming languages. One of us had never touched `Virgil` before this project, and the other had worked solely in the backend of the compiler, with no experience with the intermediate phases and associated classes.

The one thing we missed the most was a language server that would allow us to go to the definitions of classes and methods without needed to globally search for them in the project directory. Since there was virtually no documentation for the compiler, we relied heavily on jumping around and reading both the definitions of various classes/methods as well as where they were used to put everything in context. We heavily relied on Ben Titzer's knowledge and expertise of the compiler to guide us when we were stuck.

VI. Conclusion

We present the design, implementation, and analysis of subtype specialization of methods for the `Virgil` compiler. This optimization involves generating duplicate versions of methods that take in a more specific

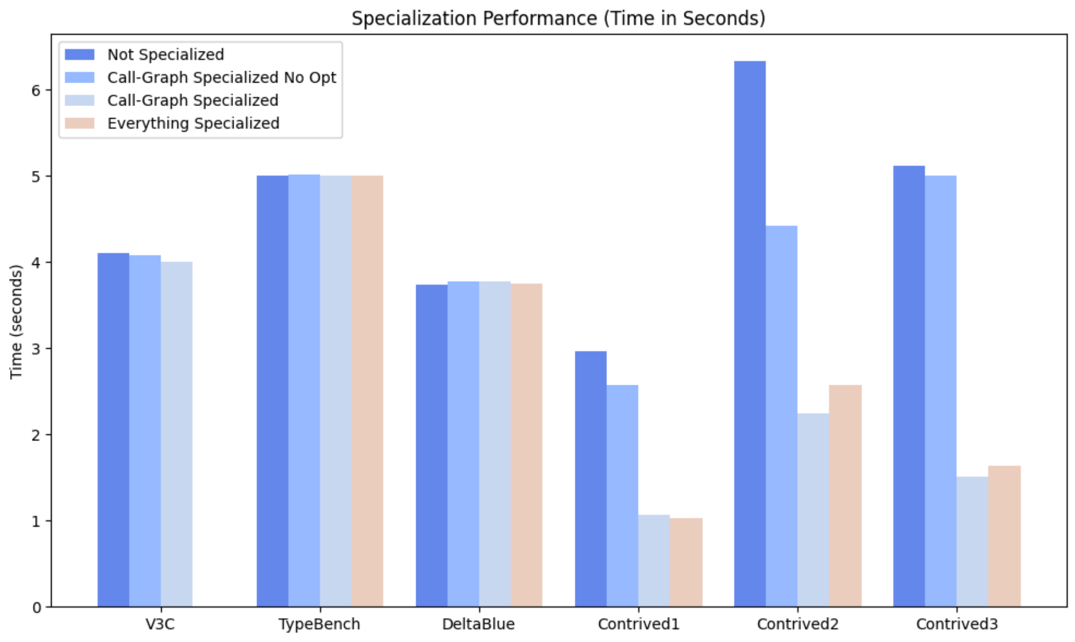

Figure 1: Chart comparing runtime performance of subtype specialization optimizations

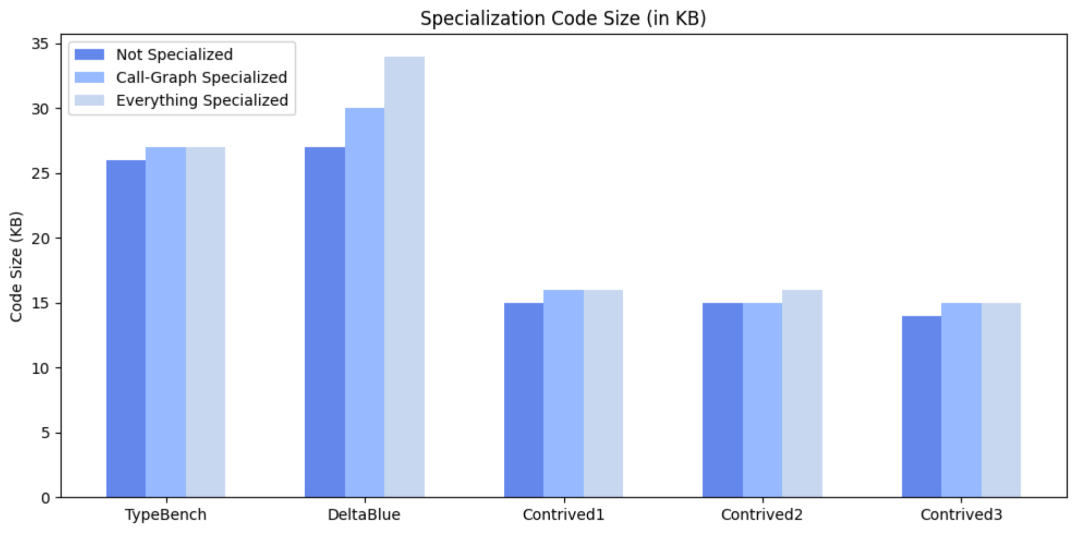

Figure 2: Chart comparing generated binary sizes for benchmark programs

subtype of their parameters, potentially devirtualizing class method calls for that specialized parameter. This allows for more optimizations, such as inlining, to be more effective on the specialized method.

We demonstrate significant increases in runtime performance for our contrived microbenchmarks, but observe no meaningful increase for more realistic benchmarks. We also show a minimal increase in generated binary size.

VII. Future Work

The optimizations we performed for this project can be greatly extended for more scenarios and analysis. We present a few directions we could have further explored.

VII.A. Specializing Class Immutable Members

We chose to specialize methods to more specific parameter subtypes because it was a simple place to start and this type of polymorhpism exists frequently. Another approach that we intended to implement, but did not because we ran out of time, is specializing class immutable members. Some design patterns, such as the visitor design pattern, involve immutable, polymorphic class members. If you specialize a class by replacing a polymorphic class member with a more specific subtype, you can eliminate all virtual dispatches involving that member in all methods of the class.

VII.B. Class Type Queries At Runtime to Call Specialized Methods

In our final implementation, we chose to only specialize methods parameters where there exists a call-site where the compiler can statically determine that a more specific subtype will be passed in. This is simple, but drastically limits the opportunities for specialization, because most of the time the compiler cannot statically determine the subtype of an object.

To increase the power of specialization, we could insert type queries at callsites where we do not know the type of the object. After determining the exact type, we could call into a specialized method. The problem with this approach is that there will be significantly more opportunities to specialize, and not all of them will be beneficial. The compiler will have to employ more complicated heuristics to decide when these specialization opportunites are beneficial.

VIII. Distribution of Work

We both contributed equally.

Bibliography

[1] J. Bezanson *et al.*, “Julia: dynamism and performance reconciled by design,” *Proc. ACM Pro-*

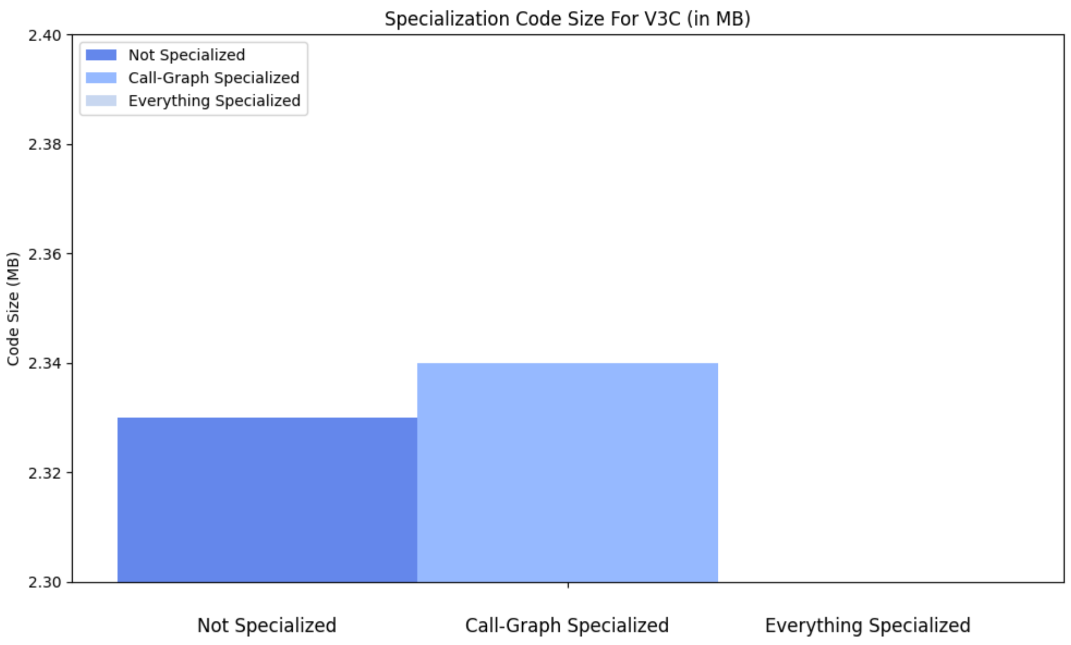

Figure 3: Chart comparing generated binary sizes for the `Virgil` compiler

gram. Lang., vol. 2, no. OOPSLA, Oct. 2018, doi: 10.1145/3276490.

[2] C. Chambers and D. Ungar, "Customization: optimizing compiler technology for SELF, a dynamically-typed object-oriented programming language," in *Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation*, in PLDI '89. Portland, Oregon, USA: Association for Computing Machinery, 1989, pp. 146–160. doi: 10.1145/73141.74831.

[3] C. Chambers and D. Ungar, "Customization: optimizing compiler technology for SELF, a dynamically-typed object-oriented programming language," *SIGPLAN Not.*, vol. 24, no. 7, pp. 146–160, Jun. 1989, doi: 10.1145/74818.74831.

IX. Tables

IX.A. Specialization Performance

	V3C	TypeBench	DeltaBlue	Contrived1	Contrived2	Contrived3
Not Specialized	4.10	5.00	3.74	2.97	6.33	5.12
Call-Graph Specialized No Opt	4.08	5.02	3.78	2.58	4.42	5.00
Call-Graph Specialized	4.00	5.00	3.78	1.07	2.25	1.51
Everything Specialized	-	5.00	3.75	1.03	2.57	1.64

Table 1: Specialization Speedup

IX.B. Specialization Code Size

	V3C	TypeBench	DeltaBlue	Contrived1	Contrived2	Contrived3
Not Specialized	2.33M	26K	27K	15K	15K	14K
Call-Graph Specialized	2.34M	27K	30K	16K	15K	15K
Everything Specialized	-	27K	34K	16K	16K	15K

Table 2: Specialization Code Size