~~Bachelor Thesis~~ Research Proposal

A zero-cost dependency injection system?

Author: Tiziano Coroneo [2736905]

1st supervisor:	Atze van der Ploeg
daily supervisor:	Atze van der Ploeg
2nd reader:	TBD

~~A thesis submitted in fulfillment of the requirements for the VU Bachelor of Science degree in Computer Science~~

June 6, 2024

Introduction

Consider the following piece of Swift 5.10 code:

```swift
class APIClient {
 init() {}
 func request(url: String) async throws -> String { /*[...]*/ }
}

class CheckoutService {
 let apiClient = APIClient()
 init() {}
 func performCheckout() async throws {
 let response = try await apiClient.request(url: "https://example.com/checkout")
 print("Data fetched successfully! Response: \(response)")
 }
}
```

Instances of the class `CheckoutService` need to use the services offered by `APIClient`. In this implementation, during the initialization of `CheckoutService` we create an instance of `APIClient` out of the blue and use its `request(url:)` method to perform the checkout by contacting an unspecified backend API.

If we need an instance of `CheckoutService` to perform the checkout operation, we can instantiate the class and checkout away. However, what if we have multiple classes that require a reference to the same `APIClient`? What if we need multiple instances of the `CheckoutService`, each with a slightly differently configured `APIClient`? What if we want to unit-test the `CheckoutService` class? In all these cases, you want to *inject* the specific `APIClient` instance that the `CheckoutService` *depends on*.

For example, we may pass the specific instance of the `APIClient` to use in the initializer of the `CheckoutService` class, a method called "Constructor Injection" by Martin Fowler [1]:

```swift
class CheckoutService {
 let apiClient: APIClient
 init(apiClient: APIClient) {
 self.apiClient = apiClient
 }
 func performCheckout() async throws {
 let response = try await apiClient.request(url: "https://example.com/checkout")
 print("Data fetched successfully! Response: \(response)")
 }
}

let checkout = CheckoutService(apiClient: APIClient())
```

Another option is to assign the instance to a property on the `CheckoutService` after initialization is complete, called "Setter Injection":

```swift
class CheckoutService {
 var apiClient: APIClient?
 init() {}
 func performCheckout() async throws {
 let response = try await apiClient?.request(url: "https://example.com/checkout")
 print("Data fetched successfully! Response: \(response ?? "")")
 }
}
```

```swift
let checkout = CheckoutService()
checkout.apiClient = APIClient()
```

Or even pass the instance directly to the method that requires it:

```swift
class CheckoutService {
 init() {}
 func performCheckout(apiClient: APIClient) async throws {
 let response = try await apiClient.request(url: "https://example.com/checkout")
 print("Data fetched successfully! Response: \(response)")
 }
}

let checkout = CheckoutService()
checkout.performCheckout(apiClient: APIClient())
```

These seem straightforward solutions to a simple problem: how can we decouple an object from its dependencies? As we will see, this apparent simplicity hides a world of complexity that no ambitious library author can ignore.

Modern mobile apps grow larger and larger over time, adding new features and extending existing capabilities. Some large-scale apps like Uber can have ~300 modules, amounting to ~1,000,000 lines of code [2]. As they grow, their internal structures also grow so complex that their authors spend a substantial amount of engineering in making systems to manage said complexity, be they abstract sets of principles like SOLID [3] or support systems for everyday operations.

A category of these support systems is “Dependency Injection” (DI) [1].

DI solves the problem of isolating code sections from their direct dependencies so that unit testing is as easy as possible while not overcomplicating the code base with additional functionality only meant to support testing. DI systems solve this issue by splitting building objects from using them and removing strong coupling between components [4]. They typically do so by defining an *object container* that holds the dependencies needed by our program, provides easy access to the objects within, and allows overriding objects to test different scenarios.

DI is a common feature of many ecosystems: in Android apps, `Dagger` [5], [6] is the most common library, while in `.NET` there is an integrated DI system in the `IServiceCollection` API. In iOS there is no standard, first-party solution, but many different libraries are available with different APIs, capabilities, and tradeoffs. The most popular library is `Swinject`, with ~6200 stars on GitHub, followed by Square’s `Cleanse` with 1800 stars, Uber’s `needle`[7] with 1700 stars, `Factory`[8] with 1600 stars, and the latest entry, `swift-dependencies`[9] with 1400 stars. We will also measure the performance of a dependency injection I wrote a few years ago: `Carpenter`.

Uber’s implementation is quite interesting, as it offers unique tradeoffs: while most DI systems resolve their dependency graph representation at runtime, their application is so large that they found the need to write their high-performance, compile-time safe DI system, doing their best to keep the impact on compilation times low. Their choices in designing this library offer insights into the different costs associated with dependency injection: a runtime solution is simpler to implement than a compile-time build plugin, but it may offer less type safety and impact an application’s startup time. A compile-time solution provides better safety guarantees (“if it compiles, it works!”), higher performance at runtime, but it risks slowing down the development cycle by increasing build times locally and in continuous integration.

All these frameworks provide similar capabilities while offering different performance tradeoffs. The first sub-research question explores these differences (**SQ1**):

“What are the tradeoffs in the design of a DI system?”

We will benchmark these frameworks to evaluate their performance impact in different categories compared to regular Swift code that uses no external library to manage complexity. Presenting the results constitutes the answer to the second research sub-question (**SQ2**):

“What is the performance impact of dependency injection systems in iOS?”

Then, we will benchmark the performance of the main open-source DI frameworks, discussing what it means to be a *zero-cost dependency injection system*, and providing an answer to the main research question (**MQ**):

“Is a zero-cost dependency injection system possible?”

Related work

“Zero-cost abstractions” describes some abstractions in the C++ ecosystem that follow the “zero-overhead principle,” first described by Bjarne Stroustrup [10].

Some works in the literature refer to DI systems with the more generic name of “Inversion of Control containers,” like the paper “Inversion-of-control layer” [11] by Sobernig and Zdun, which presents the pattern and one of its possible evolutions from a multi-modular architecture point of view.

While developing the `needle` library, Uber also made another library called `Poet`[2] to generate iOS projects with different module configurations. They used this code generation mechanism to generate test projects and benchmark the library’s capabilities in different environments.

Methodology

We conducted a series of benchmarks and analyses to investigate the tradeoffs and performance impact of dependency injection (DI) systems in iOS. This section outlines the procedures and criteria used to evaluate the frameworks.

We developed a codegen tool that generates projects using each library under test, creating a large object graph. We integrated this tool as a build-time plugin inside a benchmark suite that measures two different procedures:

1. **How long does it take to create the object graph?**

The most extensive object graphs are initialized only once at the start of the lifetime of a mobile application. The typical app has to initialize an API client, maybe a local database, deeplink handlers, push notification handlers, some secure storage for passwords and sensitive data, and a logging system… and most of these objects are interdependent and never deallocated for the lifetime of the application. Measuring the creation of the dependency injection system's object graph captures how long it takes for each library to create this first network of objects, which, from here on, we will call the "DI container."

2. **How long does accessing all objects in the dependency injection container take?**

We must simulate object accesses representing common usage patterns. Apps make API calls, save things to databases and caches, get the current time/date/locale/location, and access all these kinds of dependencies all the time. We approximate the typical usage of the objects in the dependency graph by simulating a high number of sequential accesses to the objects in the DI container.

Is a zero-cost dependency injection system desirable?

In evaluating dependency injection (DI) systems, it's important to differentiate between the costs associated with "creating the graph" and those tied to "accessing the graph."

Creating the graph is a foundational step that usually occurs once at the application launch. Since this process is a one-time overhead, performance concerns are relatively minor as long as the initialization completes within an acceptable threshold: in waiting for an application to launch on iOS, the user will accept a few tens of milliseconds before quickly returning to cat pictures. Even if graph creation is slightly slower, it might not significantly impact the overall user experience as long as it stays within bounds. On the other hand, accessing the graph is an operation that occurs repeatedly throughout the application's lifecycle. This frequent interaction with the DI system means that the performance of accessing the graph is critical and should be optimized. The quicker and more efficiently an application can access its DI graph, the better it will perform during regular use. Therefore, when assessing or designing a DI system, emphasis should be placed on optimizing how the system handles repeated accesses to the object graph, ensuring swift and efficient retrieval and interaction with the dependencies it manages.

Benchmarking Setup

The hardware used for benchmarking is a MacBook Pro equipped with an Apple M1 chip and 16GB of RAM. Software-wise, we used macOS Sonoma 14.5, Xcode 15.4, and Swift 5.10. The project setup involves the two benchmarks, the codegen tool that generates standardized projects, and a series of "project templates" combined with the codegen tool.

We identified four key metrics to evaluate each DI system's performance impact: startup time, access time, memory usage, and instructions count.

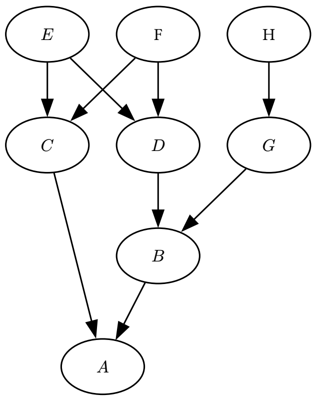
- **Startup time** is how long it takes to generate the object graph container.
- **Access time** is how long it takes to access the different objects within the container.

- **Memory usage** represents how much RAM the system uses while performing the tasks (in megabytes).
- **Instruction count** represents how many instructions the generated project executes to perform the tasks as a proxy for the final binary size.

These metrics capture the critical aspects of runtime performance. We use a recent benchmarking library called `SwiftBenchmark` [12] to capture these metrics. This library lets us record and compare measurements and quickly set up the project using the Swift Package Manager [13].

First, we need a baseline to compare to the other libraries: a project that creates the graph “by hand” without using any DI framework. A project using no external library would serve as our control to understand the overhead introduced by each DI system. This baseline framework constructs each object in the correct order, like in Listing 1, and stores all of them in a dictionary keyed by the `ObjectIdentifier` of the type of object, as shown in Listing 2. Storing the objects in such a collection allows for an easy retrieval of the built objects by type.

```
let a = A()
let b = B(a: a)
let c = C(a: a)
let d = D(b: b)
let e = E(c: c, d: d)
let f = F(c: c, d: d)
let g = G(b: b)
let h = H(g: g)
```



Listing 1: This piece of code initializes several objects in topological order. The arrows in the graph indicate a "depends on" relation.

```
[
ObjectIdentifier(A.self): a as Any,
ObjectIdentifier(B.self): b as Any,
ObjectIdentifier(C.self): c as Any,
ObjectIdentifier(D.self): d as Any,
ObjectIdentifier(E.self): e as Any,
ObjectIdentifier(F.self): f as Any,
ObjectIdentifier(G.self): g as Any,
ObjectIdentifier(H.self): h as Any
]
```

Listing 2: This is how the simple template project stores the built objects graph. ObjectIdentifier is a unique pointer to that type's metadata in the Swift runtime, guaranteeing a unique value for each type in the dictionary.

Development process

The process began by creating a small project generator to take a generic graph and generate a class for each node and a property for each edge. For an edge e from A to B, the codegen would generate the following code:

```swift
class A {
 let b: B
 init(b: B) { self.b = b }
}
class B {
 init() {}
}
```

The next step was to add more generators, one for each library, according to each library's documentation and best practices. Most libraries are very similar in their setup and operation, so that did not take long, with one glaring exception: Uber's `needle` library also contained a code generation step. Figuring out how to chain two build plugins that depend on each other in the Swift Package Manager has been an exciting debugging journey.

Implementing the benchmarks on top of the generated project took little time, as all generated projects adhere to the same protocol, so we can use polymorphism to reuse the benchmark code regardless of which library is adopted internally. Specifically, the projects implement the following protocol:

```swift
protocol GeneratedProject {
 associatedtype Container
 func makeContainer() -> Container // Create an object graph and store it in a box
 func accessAllInContainer(_ container: Container) // Access every object in the box
}
```

To perform the benchmark, we used the facilities provided by the `SwiftBenchmark` library: running the following script from the root folder of the project is enough to download dependencies, compile everything, generate the template projects, run the benchmarks, and generate a report in JMH format for later analysis:

```
swift package --allow-writing-to-package-directory benchmark --format jmh
```

There is also a Makefile with a single rule to run this script.

The report contains information for each benchmark: "Access all" and "Create container." I displayed the results using the JMH online visualizer.

We will measure each DI system's performance on a randomly generated hierarchical DAG, with characteristics determined by the "project.spec" file within each benchmark folder. This file contains the following data:

- Width: How wide should each layer of the graph be? (10-15)
- Height: How many layers should the graph have? (10, 15)
- Density: What is the probability that a node in a layer has an edge to a node in the next layer? (0.8)
- Seed: The seed of the random number generator.

We also implemented an integration with GraphViz to display an example of the magnificent graphs that will be the object of these tests:

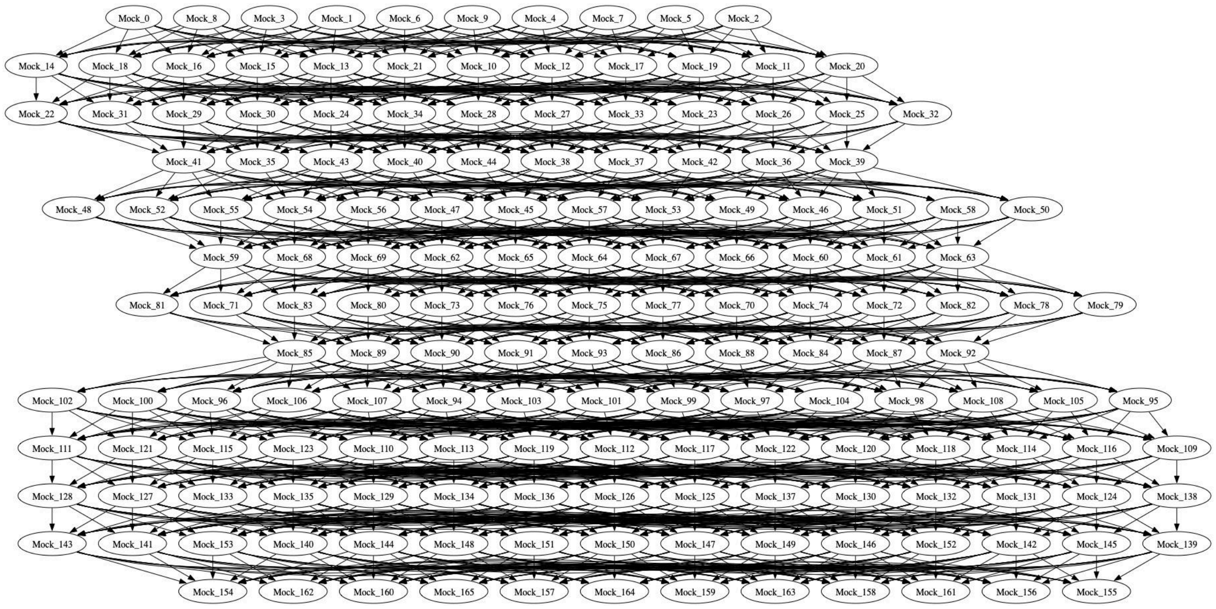


Figure 1: A somewhat realistic proxy for the typical enterprise application.

During the benchmarks' development, we analyzed the generated projects' behavior using Apple's Instruments profiling tool. We added a new target to the project (named, creatively, "Profiler target"), specifically for running the generated projects in the profiler with all optimizations. For each integration, we repeated the performance tests using Instruments.

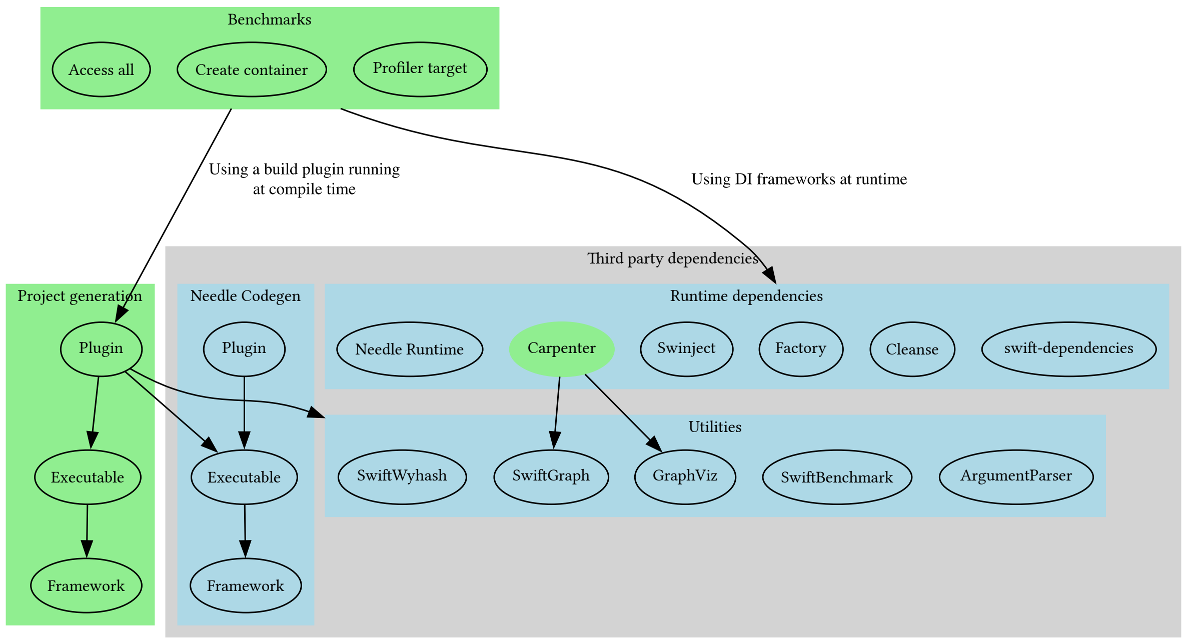


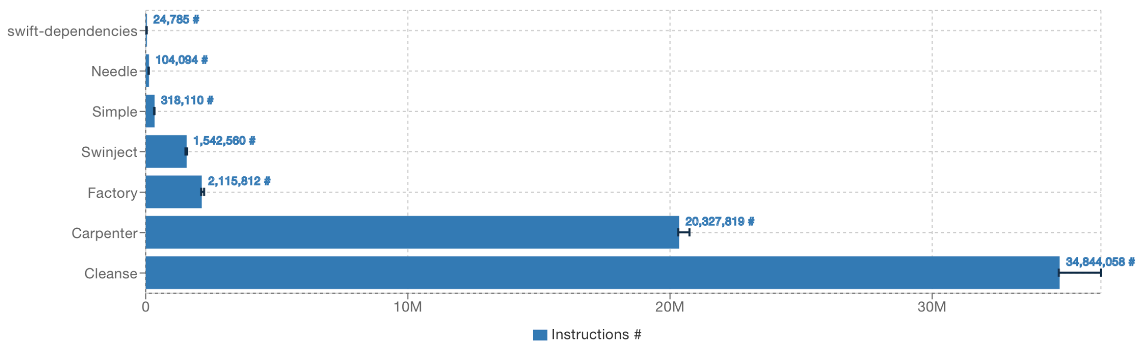
Figure 2: This is module graph of the project. The benchmarks run the project generation tool at compile time as a build plugin to create the test projects and import the various DI frameworks runtimes. The green elements represent the contributions in this paper.

Benchmark results

Creating the object graph

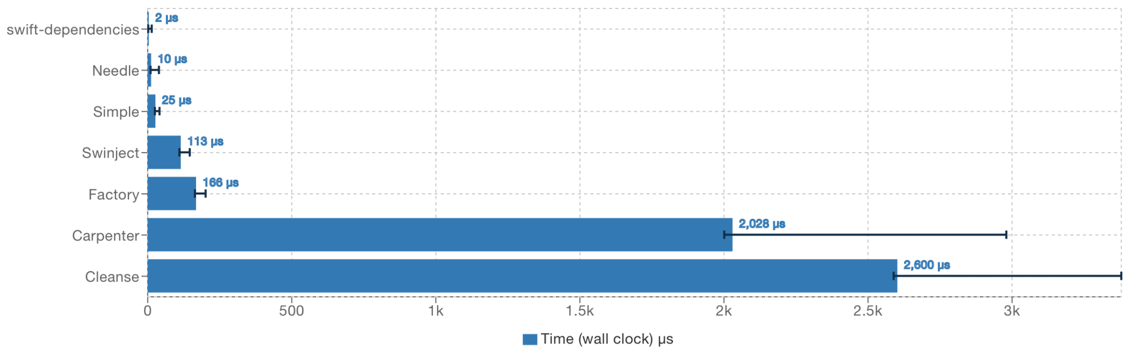
Instructions

Instructions



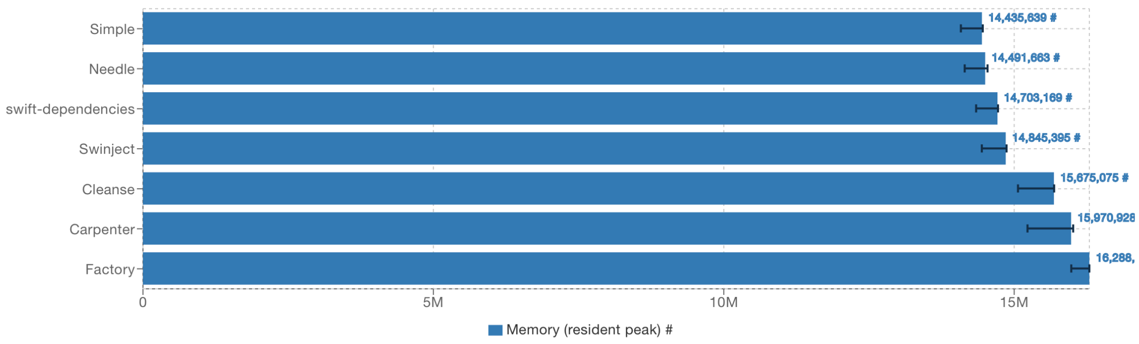
Time (wall clock)

Time (wall clock)



Memory (resident peak)

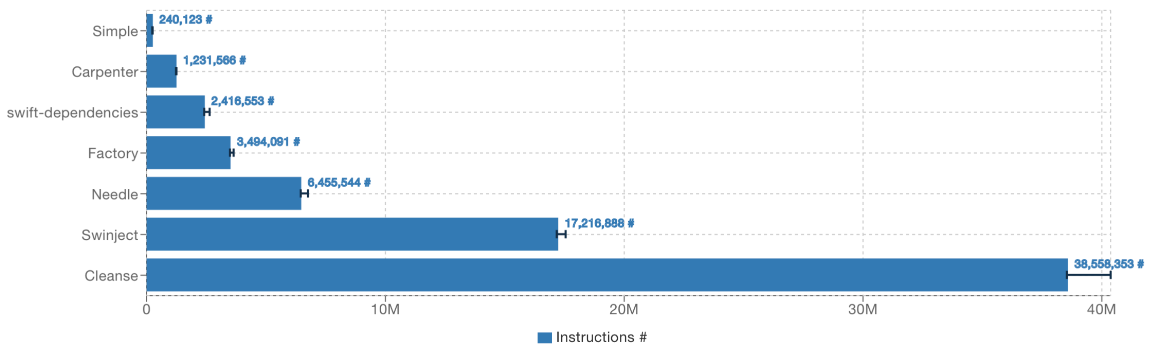
Memory (resident peak)



Object access

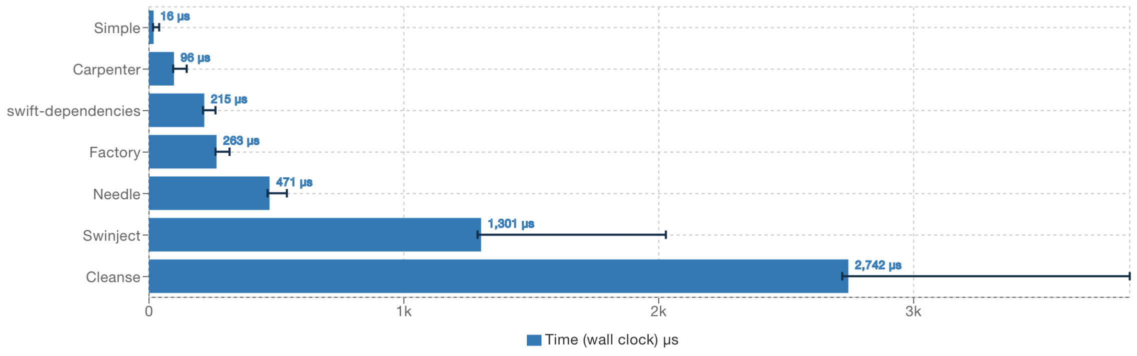
Instructions

Instructions



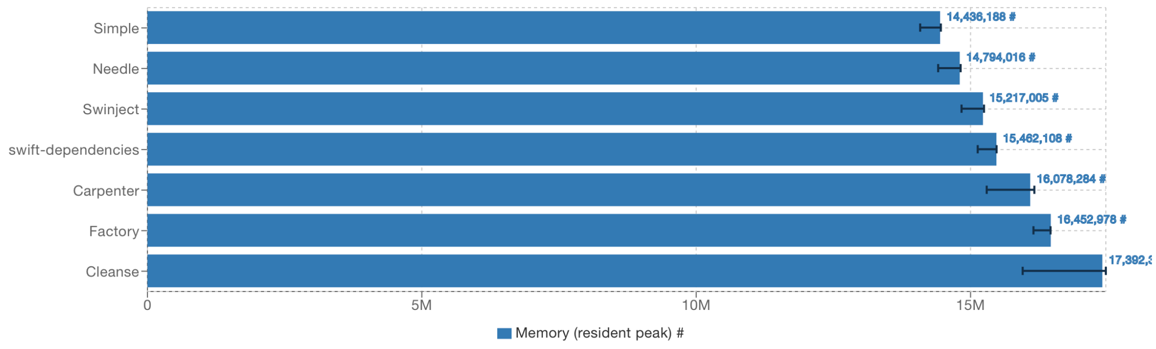
Time (wall clock)

Time (wall clock)



Memory (resident peak)

Memory (resident peak)



Results and Discussion

`Swinject` and `Cleanse`, while highly versatile and providing excellent compile-time guarantees, introduced significant runtime overhead, especially in access time. `Needle`, optimized for large-scale applications, demonstrated impressive runtime performance with minimal overhead, though it required more complex setup and integration. `Carpenter` offers a minimal set of functionality and pays the price of a relatively slow startup time to reap the benefits of fast access to the object graph. `swift-dependencies` and `Factory,` with their lightweight, functional approach, offered a good balance between simplicity and performance, while offering plenty of advanced features.

Static Initialization Impact

`swift-dependencies` and `Needle` demonstrate minimal instruction counts and suspiciously low initialization times (2 µs and 10 µs, respectively). These results are pretty baffling: How can they go faster and make the compiler emit fewer instructions than the simple implementation? In the case of `swift-dependency`, this library expects you to register dependencies as static properties on an object so that the application will initialize them once. `Needle` shows similar results for a similar reason.

This method results in high-speed access times in benchmarks but does not reflect the actual initialization cost during app launch. We discovered that this benchmarking approach needs to be revised to track the performance of code running in static initializers. How to measure the performance of the static initialization of variables is still being determined and requires further research; for this reason, we will not consider the costs of initializing the object graph when using this framework.

Memory and Dynamic Initialization Considerations

Both `Carpenter` and `Cleanse` exhibit higher instruction counts and longer initialization times (over 20 million instructions and around 2,028 µs to 2,800 µs) compared to Simple, which only requires 318,110 instructions and 25 µs. The increased costs for these systems arise from creating additional runtime representations of objects, involving extra allocations and management overhead, leading to increased memory usage and computational time. This complexity can be beneficial for applications that require additional features at the cost of performance, such as needing dynamic object configurations or sophisticated lifecycle management, in the case of `Cleanse`, or better visualization tools, developer experience, and additional facilities related to property-injection, in the case of `Carpenter`.

General Observations

The slowest system in the "Creating the graph" category only takes 2.600µs, within the acceptable range we discussed in previous sections. We could simply be working on too small a dependency graph. This poses the question: How big should the graph be in order for its construction to take enough time to block the main thread for at least one frame? Also, does this mean that all dependency injection systems are zero-cost dependency injection systems?

The access times are more interesting. The slowest library takes 2.742µs to access its build graph; this cost could contribute to app slowdowns if it needs to be paid multiple times during the application's lifetime.

Conclusion

This paper has explored the evaluation of dependency injection systems in iOS. Through benchmarking, we have shown that the hidden costs of injection systems are hidden not because of misleading documentation but because they are too small to show up on any profiler pointed at our typical iOS application. Trade-offs are unavoidable, as in the rest of the field of Computer Science. However, in this case, the differences are so minor that, as library designers, we can afford to focus on the developer experience without worrying about the runtime performance of our application in most cases. Our findings also provide a valuable tool for other iOS engineers to benchmark large-scale systems by adding new project templates to the generator.

Future Work

A different benchmarking setup is required to correctly measure the impact of frameworks such as `swift-dependencies` and `needle`. It would also be interesting to measure the impact of the various libraries on compilation time, which Uber cited as one of the leading reasons behind the development of `needle`.

Future work could explore further optimizations to the zero-cost DI system, such as integrating with SwiftUI and other modern Swift features. Additionally, extending the benchmarking to other platforms and languages could provide a more comprehensive understanding of DI systems' performance impacts.

Bibliography

[1] M. Fowler, "Inversion of control containers and the dependency injection pattern," 2004, [Online]. Available: http://www.martinfowler.com/articles/injection.html

[2] Uber, "Introducing Uber Poet, an Open Source Mock App Generator for Determining Faster Swift Builds." [Online]. Available: https://www.uber.com/en-NL/blog/uber-poet/?uclick_id=76466083-c911-4bb1-9435-322d5fe3156c

[3] R. C. Martin, *Agile software development: principles, patterns, and practices*. Prentice Hall PTR, 2003.

[4] M. Seemann, *Dependency Injection in .NET*. Manning, 2011. [Online]. Available: https://books.google.nl/books?id=lnOqcQAACAAJ

[5] Square, "Dagger 1." [Online]. Available: https://github.com/square/dagger

[6] Google, "Dagger 2." [Online]. Available: https://github.com/google/dagger

[7] Uber, "needle." [Online]. Available: https://github.com/uber/needle

[8] M. Long, "Factory." [Online]. Available: https://github.com/hmlongco/Factory

[9] S. Celis and B. Williams, "swift-dependencies." [Online]. Available: https://github.com/pointfreeco/swift-dependencies

[10] B. Stroustrup, "Foundations of C++," in *Programming Languages and Systems*, H. Seidl, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 1–25.

[11] S. Sobernig and U. Zdun, "Inversion-of-control layer," in *Proceedings of the 15th European Conference on Pattern Languages of Programs*, in EuroPLoP '10. Irsee, Germany: Association for Computing Machinery, 2010. doi: 10.1145/2328909.2328935.

[12] ordo-one, "Swift Package Benchmark." [Online]. Available: https://github.com/ordo-one/package-benchmark

[13] Apple, “Swift Package Manager.” [Online]. Available: https://www.swift.org/documentation/package-manager/