Bachelor Thesis

All dependency injection systems are zero-cost

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 17, 2024

Abstract

This paper explores the performance tradeoffs of dependency injection (DI) systems, taking as example several different frameworks popular in iOS applications, particularly investigating if they can be considered zero-cost in terms of runtime overhead. Modern mobile applications, such as Uber, involve complex structures with numerous interdependent modules, necessitating effective dependency management solutions. DI systems like Swinject, Cleanse, Needle, Carpenter, Factory, and swift-dependencies are benchmarked to measure their performance in creating and accessing dependency graphs. The benchmarks reveal that while different DI systems have varying impacts on startup times, memory usage, and runtime performance, these costs are generally negligible. Therefore, DI systems can be classified as zero-cost under typical usage conditions. This paper provides insights into the design tradeoffs of DI systems and offers a framework for benchmarking their performance, concluding that DI systems have minimal runtime costs and emphasizing the importance of optimizing for developer experience.

Contents

Introduction

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 [1]. 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 [2] or support systems for everyday operations.

Consider the following piece of Swift 5.10 code in Listing 1:

```
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)")
  }
}
```

Listing 1: Two classes, CheckoutService and APIClient, with a direct dependency of the former on the latter.

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 [3] (Listing 2).

```
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())
```

Listing 2: Example of constructor injection.

Another option is to assign the instance to a property on the `CheckoutService` after initialization is complete, called “Setter injection” (Listing 3), or even to pass the instance directly to the method that

requires it (Listing 4). Note that the `APIClient` is now required to be optional because of Swift's "definite initialization" rules: we cannot leave a field uninitialized by the end of the initializer. The compiler forces us to mark it as `Optional` to allow the value to be null in between the initialization and its assignment.

```
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 ?? "")")
  }
}

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

Listing 3: Example of setter injection.

```
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())
```

Listing 4: Example of passing the dependency as an argument.

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.

Let us consider another case: we have an `Authenticator` object that handles the secure storage of auth tokens, and we have an `APIClient` that performs authenticated requests to a server. The `Authenticator` needs a working `APIClient` to perform the requests but must be tested in isolation. Conversely, the `APIClient` needs the `Authenticator` to perform authenticated requests and refresh the tokens according to the OAuth protocol. These objects' operations are interdependent, but we cannot inject both of them into each other initializer (Listing 5).

```
let egg = Egg(chicken: ???)
let chicken = Chicken(egg: ???)
```

Listing 5: Cyclic dependency between eggs and chickens.

The solution is to inject one of the dependencies into the other in a different way. One option is to use setter injection for the `APIClient (depends on) Authenticator` edge and to keep constructor injection for the other direction (Listing 6).

```
let apiClient = APIClient()
let authenticator = Authenticator(apiClient: apiClient)
apiClient.authenticator = authenticator
```

Listing 6: Two classes, Authenticator and APIClient, with a cyclic dependency between them solved by using different types of injection.

The issue with using setter injection is the brief window between the initialization of the `Authenticator` and its assignment on `APIClient`. The client cannot reliably perform its functionality during this time because it lacks a critical dependency. When details like these accumulate across hundreds of modules and objects managed by different teams and organizations, it is very convenient to have support systems dedicated to solving these issues in a standardized, scalable way.

The support systems for this problem are called "Dependency Injection" (DI) systems [3].

DI solves the problem of isolating code sections from their direct dependencies to achieve a higher level of modularization and simplify unit testing while burdening the developer with the smallest amount of additional maintenance work. 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`.

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?"

Then, we will analyze the results, discussing what it means to be a *zero-cost dependency injection system*, arguing why all the tested systems belong to this classification, and providing an answer to the main research question (**MQ**):

> "Are all dependency injection systems zero-cost?"

Related work

"Zero-cost abstractions" describes some abstractions in the C++ ecosystem that follow the "zero-overhead principle," first described by Bjarne Stroustrup [10]. The idea is to strive to have abstractions that only bear costs at compile time but can be optimized by the compiler to have a negligible runtime cost.

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`[1] 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.

All dependency injection systems are zero-cost

We conduct a series of benchmarks and analyses to investigate the tradeoffs and performance impact of dependency injection (DI) systems in iOS. Ultimately, this investigation will show that the costs associated with DI systems are most often negligible. This section outlines the procedures and criteria used to evaluate the frameworks and come to this conclusion.

We first develop a codegen tool that generates projects using each library under test, creating a large object graph. We then integrate 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 is essential 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 designing a DI system, emphasis should be placed on optimizing how the system handles repeated accesses to the object graph, ensuring swift and efficient interaction with the dependencies it manages.

The tradeoffs

There are many tradeoffs in the design of a dependency injection system: aiming for the best runtime performance usually results in worse compilation times, while optimizing for compile time requires more information to be made available to the compiler. This information needs to come from somewhere: either from runtime, incurring runtime costs for the application, or from the developer, resulting in a worse developer experience, slower development, and a higher risk of mistakes.

For example, Uber's implementation is quite different from the other DI systems that we examined: They tend to resolve their dependency graph representation at runtime, avoiding the need for an additional build step. However, Uber's application is so large that they found the need to write their high-performance, compile-time safe DI system to provide additional safety guarantees while 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.

We synthesize this analysis in Table 1, which answers the first sub-research question (**SQ1**).

	Runtime performance	**Compilation time**	**Developer experience**	**Features**
Add build step to minimize runtime	Application runs as fast as possible	Additional build step slows down compilation	Longer iteration and CI times	Information might be unavailable in static analysis
Build graph fully at runtime	Performance loss in launch times and throughout use	Only impact is the compilation of the framework itself	Smaller impact on iteration and CI times	More information available during runtime

Table 1: Tradeoffs involved with different choices in the design space of DI systems.

Benchmarks

What do we measure

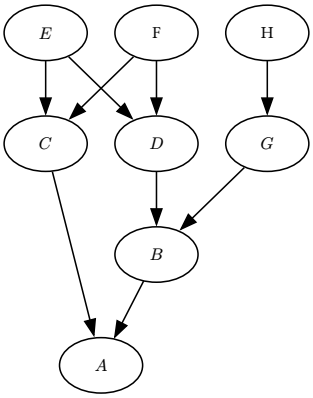
We identified four key metrics to evaluate each DI system's performance impact: startup time, access time, memory usage, and instructions count.

- **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 are generated by the compiler 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 7, and stores all of them in a dictionary keyed by the `ObjectIdentifier` of the type of object, as shown in Listing 8. 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 7: 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 8: 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.

To test DI systems, we need comparable projects that use said systems. We begin by creating a small project generator to take a generic dependency graph and generate a class for each node and a property for each edge. For an edge e from node A to node B, the codegen would generate the following code:

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

These classes represent real objects and systems that we would typically encounter in an app: database clients, API clients, deeplink handlers, global routers, app intent handling, analytics clients, all kinds of encapsulation objects around third-party vendors' code and so on.

The next step is 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 is a relatively easy endeavor, with one glaring exception: Uber's `needle` library also contains a code generation step. Figuring out how to chain two build plugins that depend on each other in the Swift Package Manager is an exciting debugging journey that we will not spoil to the reader.

After we have a reliable project generator, we can write benchmarks on top of it. Writing benchmarks is greatly simplified if all project generators conform to a standard interface that will dictate what is and is not possible within the benchmarks. Specifically, the projects implement the following protocol:

```
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
}
```

How do we measure

The hardware used for benchmarking is a MacBook Pro (2020) 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.

To perform the benchmark, we use 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
```

A `Makefile` with a single rule to run this script also exists.

The report contains information for each benchmark: "Access all" and "Create container." The results are best displayed using the JMH online visualizer [14], which produced the graphs included in this document.

We 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.

The project also contains an integration with GraphViz to display an example of the magnificent graphs that will be the object of these tests:

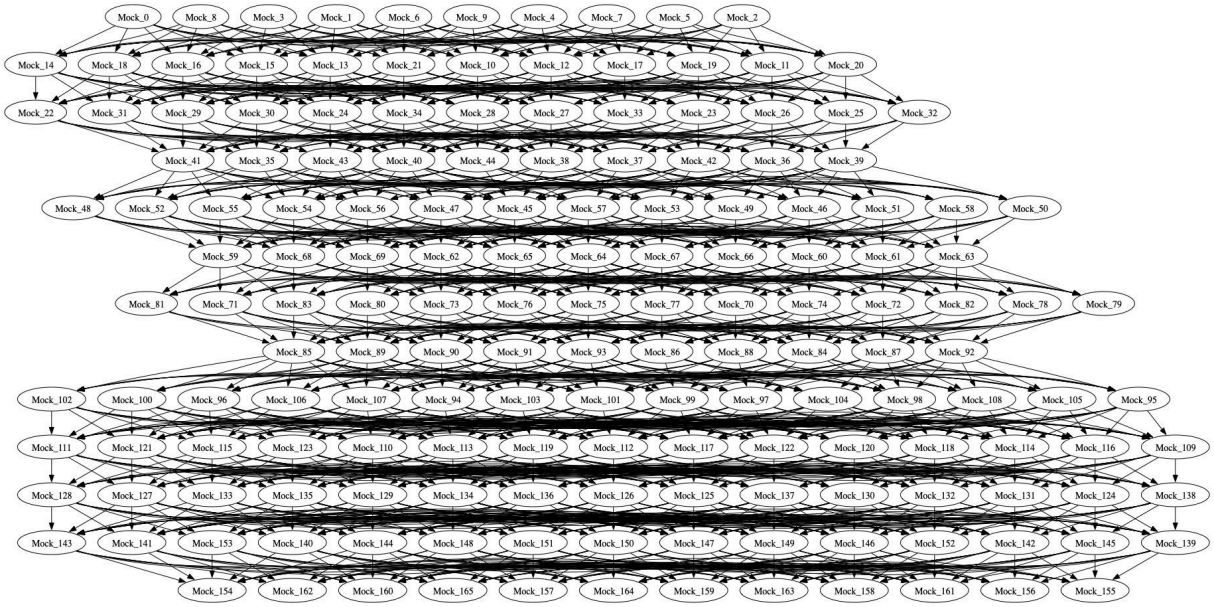


Figure 1: A somewhat realistic proxy for the typical enterprise application.

When implementing a new project generator within this benchmark setup it is good to analyze and profile the generated projects' behavior. For this, we used Apple's own Instruments profiling tool. The project includes a target (named, creatively, "Profiler target"), specifically for running the generated projects in the profiler with all optimizations enabled. For each integration, we repeated the performance tests using Instruments.

Figure 2 is an overview of the module graph of this benchmarking and project generation suite.

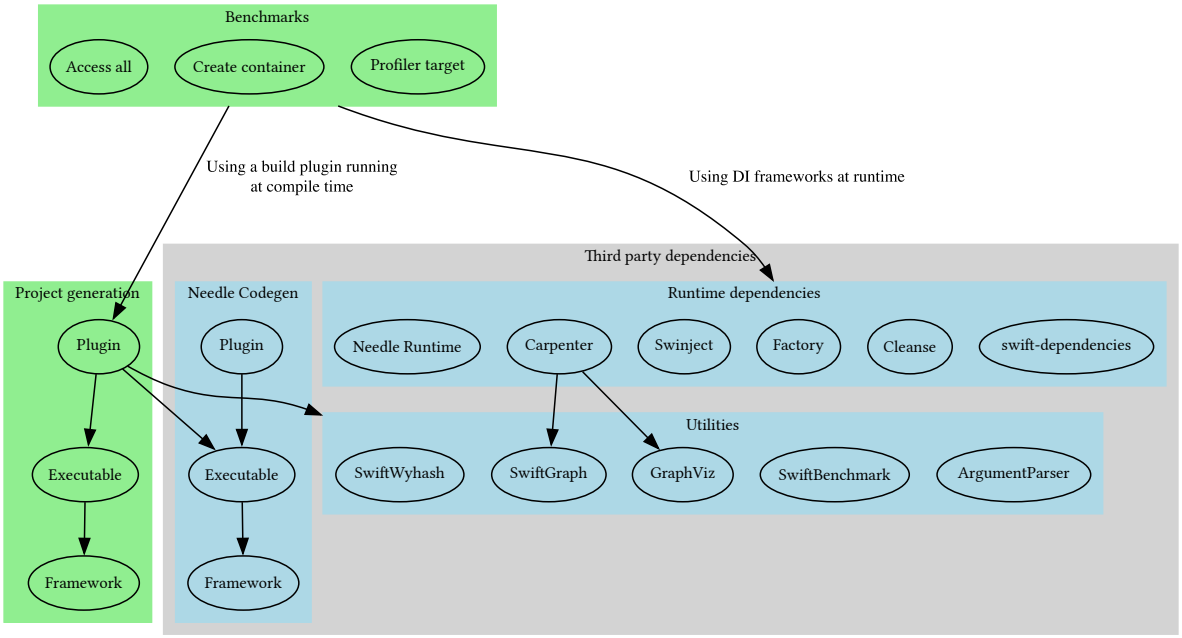
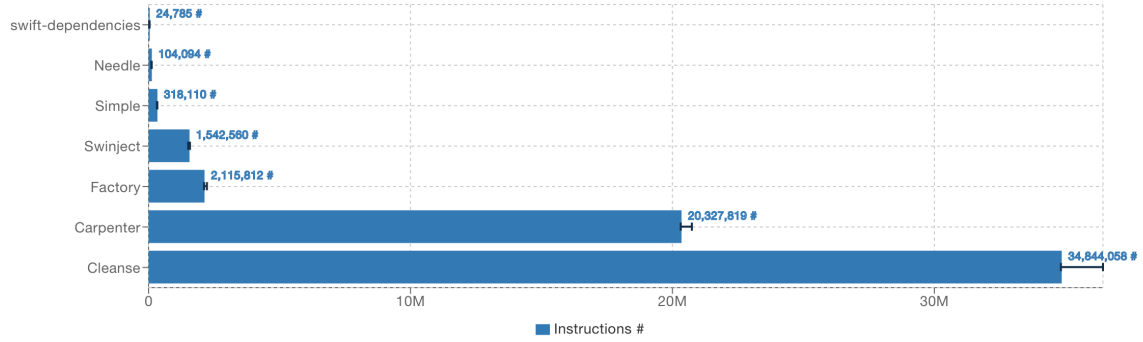


Figure 2: This is module graph of the project. The benchmarks run the project generation tool at compile time as a build plugin to create the test projects and import the various DI frameworks runtimes. The green elements represent the contributions in this paper.

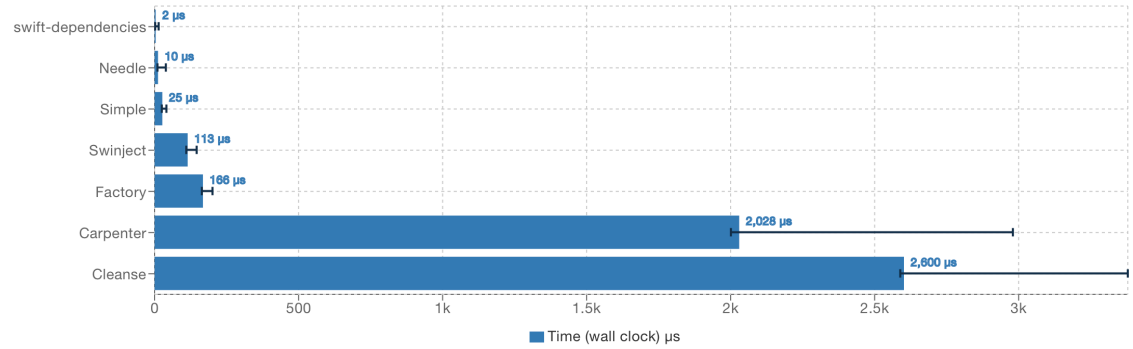
Benchmark results

Creating the object graph

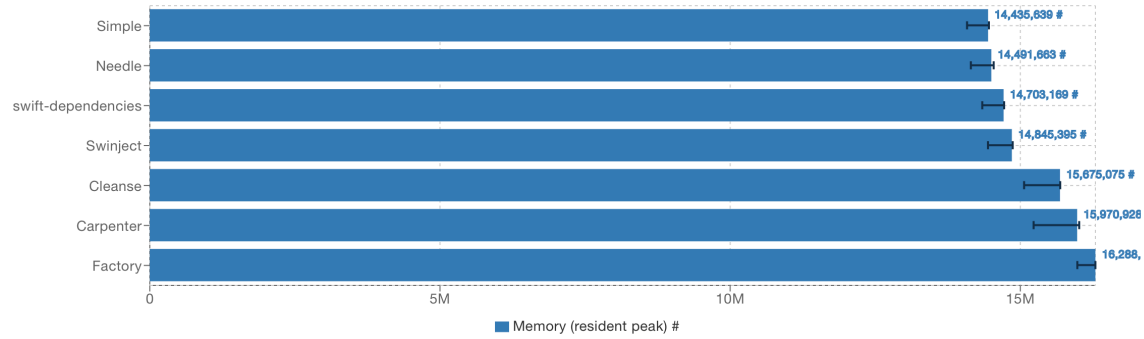
Instructions Instructions



Time (wall clock) Time (wall clock)

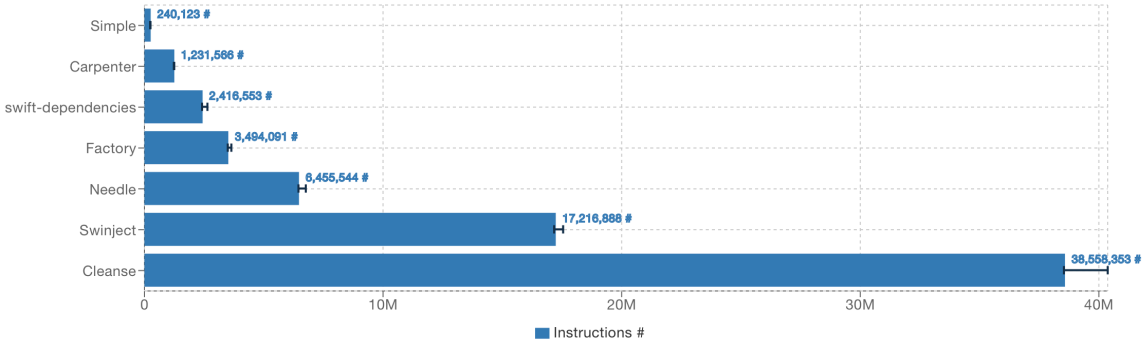


Memory (resident peak) Memory (resident peak)

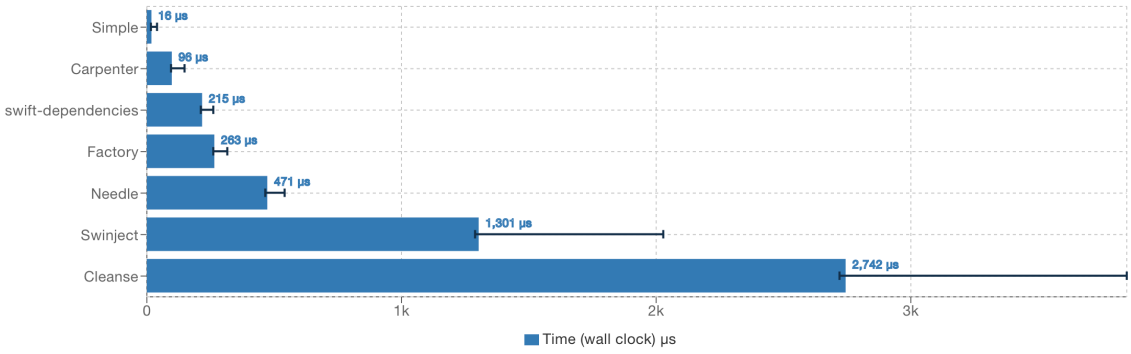


Object access

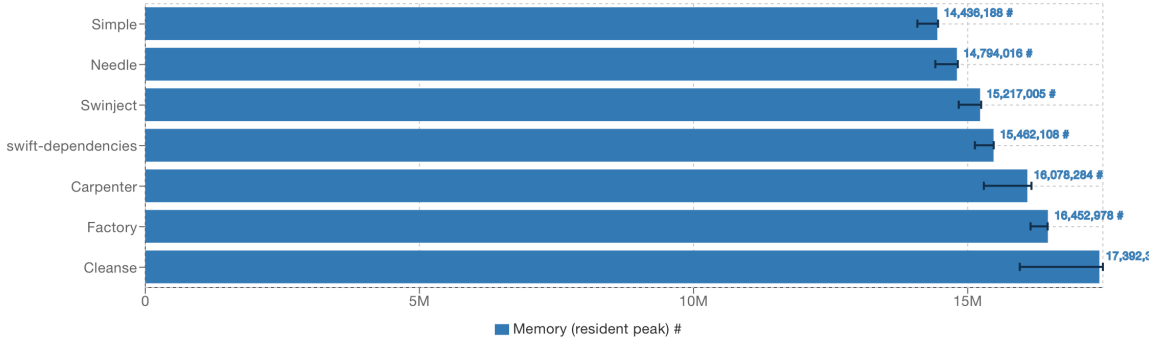
Instructions Instructions



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 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.

Do the results generalize to other platforms?

iOS devices are comparable to relatively capable embedded systems. Other ecosystems that run applications in the same class in terms of size and architecture have their own preferred method or framework for dependency injection, like the previously cited Dagger for Android, or the standard APIs in .NET: these systems run on everything from relatively low-power Android devices to the largest servers.

As this benchmark focused on one of the lowest-powered devices in this broad range, we can be confident that these results will be valid in other platforms with more computational capacity, like laptops, servers, or distributed systems. Considering other embedded devices instead, most software running on these has an entirely different set of constraints, so much so that some features of regular Swift code are completely unavailable when compiling in the recently introduced embedded compilation mode [15]. The need for complex dependency injection systems is out of place if the most pressing issue is minimizing binary size or memory consumption.

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 impact usage in any 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 our application's runtime performance in most cases. Our findings also provide a valuable tool for other developers to benchmark DI systems by adding new project templates to the generator.

> "Are all dependency injection systems zero-cost?"

Yes, the runtime costs of DI systems are always insignificant under realistic assumptions (**MQ**).

Future Work

Testing graphs created from real-world traces would give more significant insights into production applications' performance costs.

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] 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

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

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

[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/

[14] “JMH Visualizer.” [Online]. Available: https://jmh.morethan.io/

[15] Apple, “Embedded Swift.” [Online]. Available: https://github.com/swiftlang/swift-evolution/blob/main/visions/embedded-swift.md