KTH ROYAL INSTITUTE
OF TECHNOLOGY

Degree Project in Computer Science and Engineering

Second cycle, 30 credits

# Evaluating Swift concurrency on the iOS platform

## A performance analysis of the task-based concurrency model in Swift 5.5

**ANDREAS KÄRRBY**

Stockholm, Sweden, 2022

# Evaluating Swift concurrency on the iOS platform

## A performance analysis of the task-based concurrency model in Swift 5.5

ANDREAS KÄRRBY

Degree Programme in Computer Science and Engineering
Date: August 14, 2022

Supervisor: Pablo Buiras
Examiner: Mads Dam
School of Electrical Engineering and Computer Science
Host company: Bontouch AB
Swedish title: Utvärdering av Swift concurrency på iOS-plattformen
Swedish subtitle: En prestandautvärdering av den task-baserade concurrency-modellen i Swift 5.5

# Abstract

Due to limitations in hardware, raising processor clock speeds is no longer the primary way to increase computing performance. Instead, computing devices are equipped with multiple processors (they are *multi-core*) to increase performance by enabling parallel execution of code. To fully utilize all available computational power, programs need to be *concurrent*, i.e. be able to manage multiple tasks at the same time. To this end, programming languages and platforms often provide a *concurrency model* that allows developers to construct concurrent programs. These models can vary both in design and implementation.

In September of 2021, a new version of the *Swift programming language*, most commonly used to develop mobile applications on Apple's iOS platform, was released. This release introduced a new concurrency model, *Swift concurrency (SC)*, featuring e.g. structured concurrency and the `async/await` pattern.

The performance of a concurrency model is important, not the least because end users expect applications to be responsive and performant. This thesis investigates Swift's new concurrency model from a performance perspective, comparing it to a previous model, *Grand Central Dispatch* (GCD). Six benchmark applications are developed and implemented in both the GCD and the Swift concurrency models. Three of the benchmarks are focused on exercising separate parts of the models in isolation. The other three use the models to solve classical computational problems: Fibonacci numbers, N-Queens problem, and matrix multiplication.

A performance analysis is carried out to study the differences in execution time and memory consumption between the two models. The results show differences between the two models, especially in execution time, and indicate that neither model consistently outperforms the other. Finally, some possible avenues for future work are identified.

## Keywords

Task-based concurrency, Mobile, Benchmarking, Swift, iOS

# Sammanfattning

På grund av begränsningar i hårdvara går det inte längre att öka datorprestanda genom att enbart öka klockfrekvensen hos processorer. Datorer förses numera istället med flera processorer (s.k. *multi-core*) för att öka prestanda genom att möjliggöra parallell exekvering av kod. För att till fullo kunna utnyttja all tillgänglig datorkraft så måste program vara *concurrent*, det vill säga att de måste kunna hantera flera olika uppgifter samtidigt. För detta ändamål tillhandahåller programmeringsspråk och plattformar ofta en *concurrency-modell* som låter utvecklare konstruera program som är concurrent. Dessa modeller kan variera både i design och i hur de är implementerade.

I september 2021 så släpptes en ny version av programmeringsspråket *Swift*, som främst används för att utveckla mobilapplikationer på Apples iOS-plattform. Den nya versionen introducerade en ny concurrency-modell, *Swift concurrency*, med bland annat strukturerad concurrency och `async/await`-mönstret.

Prestandan i en concurrency-modell är viktig att beakta, inte minst för att användare förväntar sig att applikationer ska vara responsiva och kraftfulla. Denna studie utvärderar den nya concurrency-modellen ur ett prestandaperspektiv, och jämför den med en tidigare modell, *Grand Central Dispatch* (GCD). Sex stycken benchmark-applikationer skapas och implementeras i både GCD- och Swift concurrency-modellerna. Tre av våra benchmarks fokuserar på att utvärdera enskilda delar av modellerna var för sig. De andra tre använder modellerna för att lösa klassiska beräkningsproblem: Fibonacci-tal, N-Queens-problemet, och matrismultiplikation.

En prestandaanalys utförs för att studera skillnaderna i exekveringstid och minnesanvändning mellan de två modellerna. Resultaten visar på skillnader mellan de två modellerna, särskilt i exekveringstid, och indikerar att ingendera modell konsekvent presterar bättre än den andra. Slutligen identifieras några möjliga vägar för framtida arbete.

## Nyckelord

Task-baserad concurrency, Mobil, Benchmarking, Swift, iOS

# Acknowledgments

I would like to thank the host company, Bontouch, for giving me the opportunity to conduct my thesis project with them and for letting me take part in the day-to-day meetings and happenings at the company, which provided a nice break from the work. I also want to thank the people at Bontouch who helped me with my questions and provided advice and guidance.

I would like to especially thank my two supervisors, Sofia and Pablo, for aiding me throughout the process of this thesis. You have truly gone above and beyond what I could ever have asked for and I feel truly grateful for all our insightful discussions and all the feedback I have received.

Finally I would like to thank KTH for all these years; for everything I've learned and all the wonderful people I have met. My years at KTH will, without a doubt, be some of the most memorable of my life.

Thank you.

Stockholm, August 2022
Andreas Kärrby

# Contents

# List of Figures

# List of Tables

# Listings

# List of acronyms and abbreviations

**CPS** continuation-passing style

**GCD** Grand Central Dispatch

**JVM** Java Virtual Machine

**LLVM IR** LLVM Intermediate Representation

**QoS** quality-of-service

**SC** Swift concurrency

**SDG** Sustainable Development Goal

**SIL** Swift Intermediate Language

**UI** user interface

**WWDC** Apple Worldwide Developers Conference

# Chapter 1

# Introduction

In the past, programmers could get away with waiting a few months for better hardware to make their programs run faster (without needing to re-write those programs to fully utilize the new hardware). The programmers of today, however, need to take special care when constructing their programs, in order to take full advantage of the available hardware real estate [1]. The reason for this shift is that technology advancements in the processor-manufacturing industry no longer lead to increased clock speeds. Instead, we are moving towards more processing units ("*cores*") per CPU (see Figure 1.1). In order for a program to fully make use of these cores, it needs to be constructed in a way that allows it to manage multiple tasks at the same time (so that the work can be efficiently distributed among the available cores). In other words, it needs to have a high level of *concurrency*.

The same trend of increasing the number of cores per CPU can also be seen in smartphones. These devices are becoming more and more sophisticated, and users expect them to be able to perform increasingly complex and computationally intensive tasks [4]. Consequently, programmers need to program with concurrency in mind, and programming language designers need to provide an appropriate abstraction that helps manage the inherent complexity of concurrency.

## 1.1 Concurrency

Concurrency refers to the ability of a program to manage multiple tasks simultaneously. It is important to note that it does not imply parallelism (i.e. actually *executing* several things at the same time). It is, however, a common way to enable parallelism, because tasks that are managed simul-

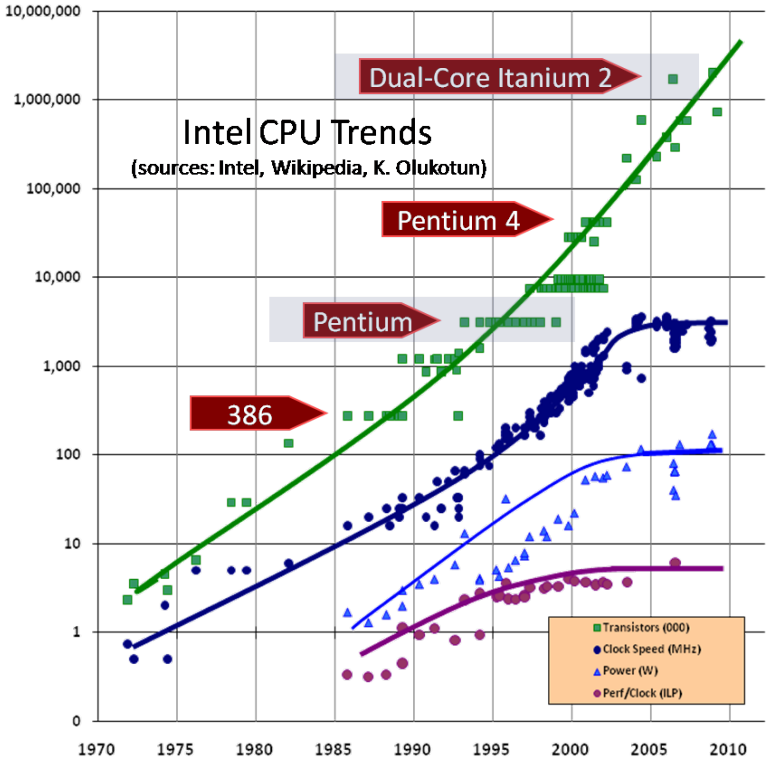



Figure 1.1: CPU characteristics over time. The number of transistors keep increasing, while clock speed, power, and performance per clock cycle flatten out around the year 2004. Image credit: [1].

taneously in a concurrent setting can easily be parallelized if the necessary hardware is available.

With that in mind, many common programming language platforms enable programmers to write concurrent programs, e.g. by way of standard libraries, or directly as built-in features of the languages themselves. One of the most common approaches is to allow the programmer to create *threads*. These threads of execution all share the same memory (they are part of the same operating system *process*) but they can independently run code from different parts of a program [5]. As an example, Java provides the *Thread API*, allowing users to create and manage threads to concurrently handle different tasks within a program; these threads can efficiently be scheduled to run in parallel on different CPU cores [6].

## 1.2 Concurrency on the iOS platform

On the *iOS platform*, which is used on Apple's smartphones, all user interactions – such as swiping, tapping a button, etc. – are handled by a "main" thread, and all user interface (UI) work, such as updating the user interface in some way, is also performed there [7]. This is strictly enforced, because the runtime prioritizes work on this thread to make sure that the user interface is as responsive as possible.

To not block the main thread, even when computationally intensive work needs to be done, other threads can be created that run concurrently with the main thread. The main thread can then hand off work to these other threads and, importantly, does not wait for the work to complete before resuming its main job of handling user interactions. This is the notion of an *asynchronous* computation, i.e. a computation that is not waited on to finish (from the perspective of the initiator of the work).

Even on a single-core system (which can only run one thread at a time), this division of work between threads allows UI work needed for a responsive user experience to be prioritized and executed in a timely manner while still time-sharing the CPU with the other threads. The fact that smartphones now have multi-core processors since around 2011 additionally means that this work can be performed at the same time, on different cores, resulting in performance improvements in terms of overall speedup [4].

However, while it is possible to create threads explicitly and manage them directly in a program, the iOS platform also provides abstractions that let programmers delegate thread management to the underlying system. These abstractions (which will be briefly explained in the two subsequent sections, and explained in more detail in Chapter 2), and their performance implications, are what constitute the main focus of this degree project.

### 1.2.1 Grand Central Dispatch (GCD)

The *Objective-C programming language*, and more recently the *Swift programming language* (both introduced in further detail in Chapter 2), are the programming languages used to develop mobile applications for Apple smartphones. Before Swift version 5.5 (released in September, 2021)[1], the main way of writing concurrent programs natively with these languages have relied on operating system support in the form of the *Grand Central Dispatch (GCD) framework* [8]. It provides programmatic access to submit – or,

[1] https://www.swift.org/blog/swift-5.5-released/

*dispatch* – work items (called *tasks* or *blocks*) to different *dispatch queues*. Tasks are dispatched to the *main queue* when the work is to be done on the main thread, and otherwise to one of the four globally available background queues. The background queues have different priorities, all lower than that of the main queue. The GCD framework manages the scheduling of these tasks to run on actual threads, and creates and destroys threads to match the current demand [8].

Dispatching a task to a queue is done by invoking either the `async()` or the `sync()` function on that queue. The `async()` function will dispatch the task *asynchronously*, which means that the call to `async()` returns as soon as the task is submitted to the queue (but not necessarily scheduled to run). The `sync()` function will instead dispatch the task *synchronously*, which means that the dispatching thread[2] is blocked until the task has run to completion.

There are two main problems with the GCD concurrency model. The first has to do with what the resulting code looks like, and the second – arguably more important one – has to do with how threads are brought up by the runtime system to service the submitted work items on a dispatch queue.

### Problem 1: Pyramid of doom

In the GCD model, an asynchronous function is a function that may return before all of its work is done (most often because the function contains at least one call to `.async`). When calling an asynchronous function in this model, the caller can supply code (via a reference to some function, or with a *closure*, further explained in the next chapter) to that function in the form of a *completion handler*. This completion handler can then be invoked with some result of the asynchronous function once the asynchronous work within has finished.

Listing 1.1, reproduced and explained further in Chapter 2, shows an example of this. As explained in the code comments of the listing, some parts of the asynchronous `foo` function (namely lines 7-8) may execute after the function has returned, which is why a completion handler (lines 20-24) is passed when invoking the function. This completion handler is invoked on line 8, i.e. after all work (even the asynchronous parts) of `foo` is complete. Note that there is nothing magical about completion handlers per se. If `foo` were defined as in Listing 1.1 but without the asynchronous dispatch (lines 6-9), the function would not be asynchronous, and lines 24 and onwards would not execute until all code in `foo` had run.

[2] I.e. the thread that calls `sync()`.

```
1  func foo(ch: @escaping (Foo) -> Void) {
2
3      // (Possibly) do some "regular" sync work
4      // ...
5
6      DispatchQueue.global().async {
7          let res: Foo = slowFunction()
8          ch(res)
9      }
10
11     // The async() function call is asynchronous,
12     // so we may (and often will) reach this point
13     // before slowFunction returns (or even before
14     // it is even invoked)
15
16     return
17 }
18
19 // Invoking foo:
20 foo { res in
21     // ...
22     // Code that uses 'res' somehow
23     // (perhaps updating the UI)
24 }
25
26 // Other code here, parts of which may be executed
27 // before the CH that was passed to foo is executed (line 8)
```

Listing 1.1: GCD-based asynchronous function `foo` invoking its completion handler with a result.

The problem with the approach of passing completion handlers arises when several nested asynchronous function calls need to be made: the resulting code will consist of a stack of nested completion handlers – a phenomenon called "pyramid of doom". Listing 1.2 shows an example of what this could look like.

The `processImageData` function takes a completion handler of type `(Image) -> Void`, which is passed to the function as it is invoked (lines 13-15). The function body of `processImageData`, in turn, invokes a new asynchronous function, `loadWebResource`, passing a completion handler to it (lines 2-10). After a few such nested steps, the completion handler passed to `dewarpAndCleanupImage` (lines 5-7) is the location where the original completion handler, `completionBlock`, is finally invoked. Chapter 2 will explain Swift syntax and GCD in more detail, and returns to Listing 1.2 to show how it can be expressed more succinctly with the newly introduced language features.

```
1  func processImageData(completionBlock: (Image) -> Void) {
2      loadWebResource("dataprofile.txt") { dataResource in
3          loadWebResource("imagedata.dat") { imageResource in
4              decodeImage(dataResource, imageResource) { imageTmp in
5                  dewarpAndCleanupImage(imageTmp) { imageResult in
6                      completionBlock(imageResult)
7                  }
8              }
9          }
10     }
11 }
12
13 processImageData { image in
14     display(image)
15 }
```

Listing 1.2: Nested asynchronous function calls leading to "pyramid of doom". Source code adapted from [3].

### Problem 2: Thread explosion

When there are many enqueued tasks on a concurrent dispatch queue (*serial* and *concurrent* dispatch queues are explained in Chapter 2), the runtime system will spawn more threads to execute these tasks. More concretely, this will happen when a thread executing one of the tasks gets blocked (e.g. waiting for a network response or getting access to a database, etc.) [9]. Dispatching large amounts of tasks that potentially block a thread can thus become

problematic, because as the number of threads grows, so does the amount of memory required to manage them. Additionally, context switching between threads incurs a performance overhead. The more threads to switch between, the larger impact this overhead will have on general program performance [10].

Given the problems above, the Swift Core Team (responsible for the strategic direction and evolution of the Swift programming language) chose to adopt a different concurrency model, which was incorporated with the release of version 5.5 of the language [11].

### 1.2.2 Swift concurrency: `async/await`, structured concurrency, and actors

With Swift 5.5, concurrency transitioned from being part of the underlying operating system platform (with the GCD framework) to being part of the language itself, adding new keywords and compiler support. The new model is called *Swift concurrency*, and the changes can be grouped into three main parts:

- The keywords `async` and `await` are introduced [3]. The `async` keyword marks a function as asynchronous, and the `await` keyword marks a call to such an asynchronous function[3]. On their own, these keywords do not enable any concurrency.
- Added support for *structured concurrency* via `TaskGroup` and `Task`, which are contexts in which asynchronous code and functions execute. Tasks can have child tasks which are lexically scoped to their parent (meaning that their source code is enclosed by that of their parent), and whose lifetimes are constrained such that no child task outlives its parent task. Importantly, different tasks can be executed concurrently with each other, and are therefore what enable concurrent execution of asynchronous code [12].
- The addition of *actors* is meant to provide a way to synchronize access to shared data between concurrently executing tasks, to prevent data races [13].

[3] In Chapter 2, the meaning and use of these two keywords is refined further, and some other uses of the keywords are introduced.

## 1.3 Research question

Apart from changing the way that concurrency is expressed in Swift programs (i.e. the programmer-facing abstractions), the changes in the concurrency model also influence how the programs are executed on the underlying system. It would therefore be of interest to explore the consequences, in terms of performance impact, that these changes bring.

The research question of this thesis is the following:

*What differences in execution time and memory consumption can be observed when running implementations of common benchmarks based on the GCD and Swift concurrency models?*

### 1.3.1 Objectives

In order to answer the research question, the following objectives will have to be fulfilled:

- determine which benchmark programs to develop (considering factors such as their scope, category, level of granularity, etc.)
- implement the selected benchmark programs in both concurrency models
- determine a research methodology (experimental design and subsequent data analysis) for how to measure execution time and memory consumption and analyse the results
- carry out the research methodology such that empirical execution time and memory consumption measurements are obtained and analyse the differences between these measurements for the two concurrency models

## 1.4 Purpose

The purpose of this thesis can be split into two aspects. Firstly, it is of interest to the host company (Bontouch) to learn more about what new possibilities, advantages, and drawbacks that the new concurrency model of the language brings, since such information can be used to make informed decisions with regards to the adoption strategy of the new changes. Performance considerations constitute a key part of this, because ultimately the language

has to be not only pleasant to work with, but also result in programs that are efficient and performant when running.

Secondly, it is of academical interest to gain insight into how differences in programming language design and implementation affect the final performance of programs constructed by that language. The ideas that underpin the changes made to Swift in version 5.5 are being adopted by several other languages as well, and have been since a few years back. Hopefully, this thesis can contribute to the knowledge on what the implications of this might be.

### 1.4.1 Ethical issues and sustainability

To the best of the author's knowledge, there are no ethical issues posed by this degree project. The project can, however, be related to sustainability. Smartphones in general are interesting environments for programming, since they demand a much greater need for energy efficiency (to ensure good battery life, mainly). If there are differences in energy consumption between the concurrency models, it could have direct impact on sustainability. Performance improvements could lead to increased desire to adopt the new technology, which could be at the cost of sustainability in case the energy consumption also increases. These issues are further discussed in Section 6.2.

## 1.5 Delimitations

The following delimitations were chosen in order to bound the scope of the thesis, both in the interest of time and also to provide a clearly defined research objective.

1. Comparisons are made only on one single device. The hardware specification of this device is detailed in Table 3.2 in Section 3.2.3.

2. Comparisons are only made between identical "runtime environments" (operating system version, battery level, build configuration, etc.), also detailed in Section 3.2.3.

3. Only one version of the Swift programming language is used (version 5.5). Since the release of the new Swift concurrency model is relatively recent, new initiatives and projects still appear frequently, such as for example *Swift Distributed Actors* [14] and *Async Algorithms* [15]. While these projects would be interesting to consider, this thesis only investigates a snapshot of the language "as is" at the point of version 5.5.

4. Only a core subset of the concurrency models are investigated. There are APIs and constructs which merit further study, e.g. `concurrentPerform` [16] in GCD, and detached tasks [17] and task cancellation [18] in Swift concurrency, but investigating all details of the entirety of both models would not be possible within the time frame of the thesis.

5. Only execution time and memory consumption are included as metrics. While it would have been interesting to also include metrics such as energy consumption or number of threads created, measuring these was found to be too demanding within the scope of the project, so they are omitted to allow more detail for the two chosen metrics.

## 1.6 Structure of the thesis

Chapter 2 presents relevant background information about the Swift programming language, and how concurrency was managed with GCD before version 5.5, as well as how it is managed with Swift concurrency in version 5.5 and onwards. Theory and concepts of benchmarking and statistical analysis of results is briefly covered, and finally the chapter presents some related work. Chapter 3 presents the method used to solve the problem. The results of the benchmark measurements are presented in Chapter 4, and are analysed and discussed in Chapter 5. Chapter 6 concludes the thesis, and provides suggestions for future work.

# Chapter 2

# Background

This chapter provides general background information about the Swift programming language and its syntax, followed by a detailed description of both the old concurrency model (GCD) and the new one (Swift concurrency, with `async/await`, structured concurrency, and actors). Additionally, the chapter explains theory on benchmarking, as well as statistical concepts such as mean, standard deviation, and confidence intervals, which will be necessary to understand the data analysis process (Section 3.3). The chapter concludes with a brief review of related work.

## 2.1 The Swift programming language

Swift is a statically typed, compiled, general-purpose programming language developed by Apple Inc. It was released in 2014 and was made open-source in late 2015 [19, 20]. Targeting all of the Apple platforms (iOS, macOS, watchOS, and tvOS), it is meant to replace the Objective-C language, which was previously the de facto standard for developing applications on these platforms [21, 22]. In a 2019 developer survey, more than 50% of developers who used Swift stated that they used "Swift only" (as opposed to "Swift and Objective-C"), and the most popular Apple platform it was used for was the iOS platform [23].

According to Apple, the goal of the language is for it to be "the result of the latest research on programming languages, combined with decades of experience building Apple platforms" [24]. It is designed to be a safe language, with e.g. static checking for integer overflow, strict `nil`-checking, *automatic reference counting (ARC)* for memory management, and extensive use of value types [22, 24, 25].

To compile the language, the Swift compiler uses the *LLVM toolchain*[1]. In short, code is compiled to Swift Intermediate Language (SIL), optimized, and then further compiled into LLVM Intermediate Representation (LLVM IR). It is then optimized further by LLVM, and finally the machine code is generated [26, 27].

### 2.1.1 Swift syntax

This subsection explains some language features that will be necessary to understand the code examples of this thesis.

#### Argument labels and parameter names

Functions in Swift are often defined with *argument labels*, which must be specified when the corresponding arguments are passed to a function. Listing 2.1 exemplifies their usage.

#### Closure expressions and trailing closure syntax

A *closure* in Swift is a piece of code that can be passed around and be executed at a later time. It also captures references to variables and constants from the surrounding context where it was defined [28]. Among other things, closures can be used as arguments to functions such as `map` or `filter` (see Listing 2.2).

Additionally, Swift allows for *trailing closure syntax*, which means that when a function takes a closure as its last argument, that closure can be specified outside the parentheses of the function invocation. When a function *only* takes a closure as its argument, the parentheses may be omitted entirely. This, along with type inference, allows the code of Listing 2.2 to be expressed more succinctly, as in Listing 2.3. This syntax is frequently used when passing *completion handlers* to asynchronous functions, which will be discussed in Section 2.3.

#### The `@escaping` annotation

The `@escaping` annotation is used for completion handler parameters in asynchronous function declarations in the GCD model, where the completion handler is potentially invoked after the function has returned (this is explained further in Section 2.3). Listing 2.4 shows an example of this.

[1] `https://llvm.org/`

```swift
// 'outer' is the argument label,
// 'inner' is the parameter name
func foo(outer inner: String) -> Void {
    // Within function:
    // Use 'inner' to refer to parameter
}

// Outside function:
// Use 'outer' to refer to argument
foo(outer: "Hello")

// By default, the argument label is
// the same as the parameter name.
// Equivalent to `name name: String`:
func greet(name: String) -> Void {
    print("Hello \(name)!")
}

greet("John") // Error
greet(name: "John") // OK

// Use _ to omit argument label:
func greet(_ name: String) -> Void {
    print("Hello \(name)!")
}

greet(name: "John") // Error
greet("John") // OK
```

Listing 2.1: Argument labels in Swift

```swift
let numbers = [1,2,3]
numbers.map({ (number) -> Int in
    2*number
})

// result: [2,4,6]
```

Listing 2.2: Passing a closure to the higher-order function `map`.

```swift
let numbers = [1,2,3]
numbers.map { number in 2*number }

// result: [2,4,6]
```

Listing 2.3: Passing a closure to the higher-order function `map`, using trailing closure syntax and type inference.

```
1 func foo(ch: @escaping (Foo) -> Void) {
2     DispatchQueue.global().async {
3         let res: Foo = slowFunction()
4         ch(res)
5     }
6 }
```

Listing 2.4: Using the `@escaping` annotation on a completion handler parameter.

The completion handler argument, `ch`, that is passed to the function is only invoked once the block submitted to the dispatch queue (lines 2-5) is actually scheduled to run, which might be after the `foo()` function has returned. Because of this, the `@escaping` annotation is needed; it explicitly signifies that the closure can *escape* the function [28]. The exact details of why this is necessary is not relevant for this thesis, but essentially it has to do with the fact that Swift's memory management is based on *reference counting* (like smart pointers in C++, for example), which requires some care from the programmer so as not to introduce *retain cycles*, which are discardable portions of memory that are not deallocated because there is a cycle of references between them, preventing the reference counts from reaching zero. These can appear because of closures' ability to capture references to symbols in their surrounding context.

## 2.2 Concurrency in Swift

Before Swift version 5.5, which was released in September 2021 [29], concurrency was not built into the language. As is mentioned in the "Swift Concurrency Manifesto", this was a conscious decision from the start, because the language designers "specifically did not want to cut off any future directions" [30]. While the target platforms for Swift – mainly iOS and macOS – did (and still do) provide libraries for interacting directly with threads [31], developers were instead encouraged to use the Grand Central Dispatch (GCD) framework, to abstract away the intricacies of managing threads manually, instead letting the system handle them [8].

As mentioned in Chapter 1, the problems with this approach have caused the Swift Core team to adopt a new model to handle concurrency, which builds on the paradigm of structured concurrency and `async/await` [11]. The following sections describe the two models in more detail.

## 2.3 Grand Central Dispatch (GCD)

The GCD framework is based on the concept of *dispatch queues*, onto which a programmer can enqueue blocks of code (called *blocks* or *tasks*) which are then scheduled by the system to run on actual threads [32].

A queue can either be *serial*, or *concurrent*. Tasks on a serial queue are dequeued and scheduled to run in the same order in which they were enqueued (*first-in-first-out*, FIFO), and this is done in a sequential fashion (the next item on the queue is not dequeued until the preceding task has finished executing). Tasks on a concurrent queue are also dequeued and scheduled to run in FIFO order, but new tasks can be dequeued and scheduled even before preceding tasks have finished [33].

There are five different queues that are automatically available in a program. The main queue, which is a serial queue, will schedule all of its tasks to run on the main thread, which is the "base" thread of the program; it is created when the program is started, and persists until the program terminates. As mentioned in Chapter 1, work on this thread is prioritized to ensure responsiveness, because it is responsible for executing all UI work.

Apart from the main queue, there are four globally available concurrent queues with different quality-of-service (QoS)[2]. Each of these concurrent background queues is associated with its own collection of threads that perform the enqueued work. The system will create new threads[3] – or destroy idle threads – in these thread pools to match the current demand of the application [36]. This is the root cause of Problem 2 ("Thread explosion") mentioned in Section 1.2.1.

Additionally, it is possible to manually create new dispatch queues, called *private dispatch queues*, which are by default serial but can be configured to be concurrent. In the GCD model of concurrency, private serial queues are often employed instead of using locks when synchronizing access to a shared resource [11, 33].

From the programmer's perspective, enqueueing a task to one of the existing dispatch queues is done by accessing the `DispatchQueue` class and passing a closure (representing the task) to the queue with the desired QoS. If no QoS is specified, a 'default' QoS value is inferred.

[2] Roughly speaking, a higher QoS indicates higher priority to finish a task.

[3] Although it could not be found in any official documentation, various online sources (and the runtime behaviour of one of the benchmarks, explained further in Chapters 4 and 5) indicate that there is an upper limit of at most 64 threads that can be created in any such thread pool [34, 35].

```
// Enqueue work asynchronously onto
// the (serial) main queue
DispatchQueue.main.async {
    // work to be done
}

// Enqueue work synchronously onto one
// of the four background queues
DispatchQueue.global(qos: .userInitiated).sync {
    // work to be done
}

// Enqueue work asynchronously onto one
// of the four background queues
// (with qos: .default inferred)
DispatchQueue.global().async {
    // work to be done
}
```

Listing 2.5: Enqueueing work to be done on dispatch queues.

It is up to the programmer to decide if a task should be enqueued *synchronously* (which means blocking the local control flow until the task has completed) or *asynchronously* (which means returning immediately from the call as soon as the task is scheduled, but not necessarily before it has completed). Importantly, synchronously dispatching a task will not only block local control flow, but also block the thread executing the dispatchment. This is one key difference between the GCD concurrency model, and the Swift concurrency model (this difference will be further explained in subsequent sections).

Listing 2.5 shows examples of enqueueing work both synchronously and asynchronously to the various dispatch queues available (note the trailing closure syntax, described in Section 2.1.1).

Writing concurrent code with the GCD framework is then a matter of dispatching a task asynchronously to one of the dispatch queues, thus freeing up the current thread to do other work while the underlying system manages the scheduling and execution of the task.

In order to do something once a task is finished, functions that perform asynchronous work in the GCD model accept a parameter known as a *completion handler*. This completion handler is later invoked with the result of the asynchronous task, once it has finished (it can also be invoked with `Void`, as a notification mechanism to simply signal that the task is completed).

```
1  func foo(ch: @escaping (Foo) -> Void) {
2
3      // (Possibly) do some "regular" sync work
4      // ...
5
6      DispatchQueue.global().async {
7          let res: Foo = slowFunction()
8          ch(res)
9      }
10
11     // The async() function call is asynchronous,
12     // so we may (and often will) reach this point
13     // before slowFunction returns (or even before
14     // it is even invoked)
15
16     return
17 }
```

Listing 2.6: Asynchronous function invoking its completion handler with a result.

```
1  foo { res in
2      // ...
3      // Code that uses 'res' somehow
4      // (perhaps updating the UI)
5  }
6
7  // Other code here, parts of which may be executed
8  // before the CH that was passed to foo is executed
9  // (line 8 in Listing 2.6)
```

Listing 2.7: Calling an asynchronous function with a completion handler.

Listing 1.1 in Chapter 1, reproduced here as Listing 2.6, shows an example of what this could look like. Note that `foo()` will likely return before the `slowFunction()` call returns, but the caller can supply a completion handler (often in the form of a closure) when invoking `foo()`, containing code to run once `slowFunction()` has returned. Also note that, because the completion handler is invoked after `foo()` returns, the parameter must be marked as `@escaping` (as explained in Section 2.1.1).

Listing 2.7 shows how the `foo()` function of Listing 2.6 could be invoked (notice the trailing closure syntax once again).

### 2.3.1 Dispatch groups

The GCD concurrency model also supports the notion of *dispatch groups*, which allow for managing groups of tasks. When tasks are submitted to a dispatch queue, a reference to a dispatch group can be passed as a parameter, and invoking `wait()` on the group will block execution (blocking the thread that invokes `wait()`) until all tasks associated with the group have finished. Listing 2.8 shows an example of what this can look like.

```
let group = DispatchGroup()
for i in 1...numberOfTasks {
    DispatchQueue.global().async(group: group) {
        // ... work to be done
        doSomeWork(i)
    }
}
group.wait()

// Code here will not execute until all
// tasks submitted to the group have been
// scheduled and run to completion
```

Listing 2.8: Using the `DispatchGroup` class to manage a group of tasks.

### 2.3.2 Continuation-passing style (CPS)

The pattern of passing a completion handler to a function – so that it may be invoked at a later time – is related to the notion of *continuation-passing style (CPS)* [37]. In CPS, each function is passed (along with the rest of its arguments) a *continuation*, which itself is a function and which represents the rest of the program to be executed after the invoked function is finished. In this style, values are returned from a function by invoking the continuation with the return value of the function, like with `ch(res)` in Listing 2.6. This is in contrast to *direct style*, where values are usually returned to the function's caller via a `return` keyword.

The concept of CPS goes back to the 1960s, where it was originally used as an internal intermediate representation within compilers [38], since it could be employed to express jumps (`goto` statements) and labels. In "The discoveries of continuations", John C. Reynolds gives an historical account of continuations and some of their uses [38].

It is worth noting that all programming languages implicitly manage continuations, as the concept itself simply refers to "the rest of the program to be executed". For example, the continuation of a `return` statement in

a function is the code after the function invocation in the caller, branching in an if-else-statement selects one of two continuations to continue with and discards the other, and throwing an exception means discarding the current continuation and proceeding instead with the continuation representing the sequence of throws that eventually end in some `catch` (or a program crash if no such `catch` exists) [39]. Some languages – most notably Scheme and Standard ML – make continuations explicit, and provide control structures for manipulating them directly (see for example `call/cc` of Scheme and Standard ML [40, 39]).

In CPS terminology, passing a completion handler to an asynchronous function is akin to providing it with a continuation, such that two concurrent continuations co-exist; one which represents the rest of the program, following the call to the asynchronous function, and the other representing the actions to take once the asynchronous work within the function has completed.

## 2.4 Swift concurrency: `async/await`, structured concurrency, and actors

Swift version 5.5 introduces several new concepts to the language, to support the new concurrency model that the Swift Core Team envisions for the language going forward. The core idea is that instead of submitting tasks to queues and continually resizing thread pools to service these queues, the new model is based on creating only one thread per actual physical processor core, and making sure that these threads are never blocked [10].

The three major components for this new model are: the coroutine-based `async/await` keywords, structured concurrency with `Tasks` and `TaskGroup`, and *actors* for state isolation to protect against race conditions [11].

### 2.4.1 `async/await` keywords and suspension points

The `async` keyword is most commonly used to mark a function in Swift concurrency as asynchronous (there is also one other use case, which will be explained in Section 2.4.2 on structured concurrency). Asynchronous functions in Swift concurrency work differently than asynchronous functions in GCD. In GCD, a call to an asynchronous function returns immediately, potentially before all asynchronous work within (i.e. work dispatched to GCD queues) has finished. In Swift concurrency, calls to asynchronous functions

return only when all work within is complete, just like normal synchronous functions. The asynchronicity comes from the fact that an asynchronous function in Swift concurrency is a function that can yield control of its thread to the runtime system. This means that the function is temporarily suspended, while the system can use the thread to run other work. At some future point in time, the system will resume the suspended function, which then continues execution from the point where it was suspended. The function is not necessarily resumed on the same thread as it previously ran on.

The points where control is yielded are called *suspension points*. Whenever an asynchronous function is called, a suspension *may* happen, however it is not very well specified in the documentation exactly when a suspension actually happens and when it does not, so this thesis does not attempt to exhaustively list these situations. There are also other points where a suspension may occur, most notably when using constants declared with `async let` syntax, explained in Section 2.4.2.

Each point where a suspension may occur is called a *potential suspension point*, and is marked with the `await` keyword, to clearly mark points in code where execution may be suspended and interrupted by other work being performed.

Listing 2.9 shows the `async/await` version of the code previously shown in Listing 1.2 of Section 1.2.1. Instead of increasingly nested code with completion handlers, the `async/await` keywords allow sequences of asynchronous calls to be expressed in a more straight-line syntax. When `loadWebResource` is invoked on line 6, the call is marked with `await`, indicating that the `processImageData` function may be suspended there, i.e. control of the thread may be yielded to the runtime system. When the function is not suspended, the work within the invoked `loadWebResource` will run as normal, and the local control flow in `processImageData` will be blocked at the call site (line 7 does not run until `loadWebResource` has returned and its value has been assigned to `dataResource`). This is analogous to the synchronous dispatching of tasks in the GCD model, where local control flow was blocked until the dispatched task had finished. The difference is that in the new model, *only* the local control flow is blocked; the thread itself is freed up to do other work. This is what is meant by "asynchronous functions being able to give up their thread".

Put differently, even though the thread running the `processImageData` function is freed up in case a suspension occurs, the function itself is suspended and will not continue to the next line until the call has returned. The continuation of the call at line 6 is thus represented by the binding of the

```
1  func loadWebResource(_ path: String) async -> Resource
2  func decodeImage(_ r1: Resource, _ r2: Resource) async -> Image
3  func dewarpAndCleanupImage(_ i : Image) async -> Image
4
5  func processImageData() async -> Image {
6    let dataResource  = await loadWebResource("dataprofile.txt")
7    let imageResource = await loadWebResource("imagedata.dat")
8    let imageTmp      = await decodeImage(dataResource, imageResource)
9    let imageResult   = await dewarpAndCleanupImage(imageTmp)
10   return imageResult
11 }
```

Listing 2.9: Asynchronous function calls with the `await` keyword. Source code from [3].

result of the function call to the `dataResource` constant, and then the next line of code (`let imageResource = ...` at line 7), as opposed to being represented by a nested completion handler, as in the GCD approach.

### Coroutines

The `async/await` model is based on the concept of *coroutines*, which are similar to functions, except that they may yield control (be *suspended*) at any point and be resumed at a later time, continuing as if they had not been suspended in the first place [41]. In contrast to threads, which are *preemptively* scheduled, meaning it is up to the system to stop and start different threads running, this model is *non-preemptive*, or *cooperative*, since it relies on functions explicitly yielding control to the system.

Historically, different flavors of coroutines have been suggested. A. L. D. Moura and R. Ierusalimschy propose a classification of coroutines according to three criteria [40]:

- *Symmetric* vs. *asymmetric*: Symmetric coroutines can yield control to any other coroutine, whereas asymmetric coroutines (also called *semi-coroutines*) can only yield control to their caller. An example of asymmetric coroutines would be Python generators [40, 42, 43].
- *Stackful* vs. *stackless*: A stackful coroutine can yield control from within nested calls, while stackless coroutines cannot (Python generators would be classified as stackless).
- Whether or not coroutines are available as *first-class objects*, meaning they can be stored and passed around in code.

In Swift, invoking an asynchronous function with `await` does not yield control directly to another function, but instead may yield control to the runtime system. The `async/await` model of Swift could thus be classified as neither symmetric nor asymmetric, since it does not yield control to its caller (as an asymmetric coroutine would), nor is it able to yield to any arbitrary coroutine. Because `await` is only a potential yield, it could be considered either stackful (from the perspective of the initial `await` in a chain of `await`s), or stackless (from the perspective of the final `await` when execution is suspended). Indeed, a chain of `await`s may potentially never yield, in which case the asynchronous calls behave just like normal synchronous ones. Finally, Swift coroutines are not available as first-class objects. For a technical discussion of how coroutines are related to asynchronous functions in Swift, see "2021 LLVM Dev Mtg 'Asynchronous Functions in Swift"' [44].

One limitation of `async/await` is that asynchronous functions can only be called from an asynchronous context, i.e. a context where control of the thread can be yielded to the runtime system. Asynchronous functions themselves constitute such asynchronous contexts, but "normal" functions do not (their contexts are synchronous). Therefore, the `async/await` keywords alone are not enough to integrate asynchronous functions into the language. Additionally, asynchronous functions themselves do not provide concurrency. To provide concurrency, and bridge the gap between synchronous and asynchronous code, Swift introduces `Tasks` – which are contexts within which asynchronous code and functions execute – and *structured concurrency*.

### 2.4.2 Structured concurrency

The term *structured concurrency* is an allusion to *structured programming*, a paradigm which goes back to 1968 when Edsger W. Dijkstra formulated a letter to the Editor of Communications of the ACM entitled "Go to statement considered harmful", in which he argued for the discontinued use of `goto` statements in favour of control flow structures like `if`-statements and loops [45]. `goto` statements were considered to make code more complex and difficult to reason about, since it is not clear how the flow of control traverses the program, as the control can jump to anywhere in a program at any time, possibly without ever returning to where it jumped from.

Structured concurrency is about making concurrency structured and predictable much in the same way that structured programming did for general program control flow. As in structured programming, where code blocks (or *scopes*) are nested and the programmer can be certain that an outer block of

code will not complete before all of its inner blocks have completed, structured concurrency restricts the lifetime of concurrent tasks such that any *parent task* must wait for all of its *child tasks* to complete before it itself can complete.

The idea of structured concurrency itself is not new (see e.g. "Parallel Lisp Language PaiLisp and its Kernel Specification" from 1989 [46]), but it saw a revival in 2016 [47] and has, in recent years, been used to some extent: Martin Sústrik developed *libdill*, a C library for writing structured concurrent programs [48], N. J. Smith used structured concurrency in *Trio*, a Python library for asynchronous I/O [49], and Kotlin, one of the main programming languages for writing Android apps, adheres to structured concurrency to delimit the lifetime of Kotlin coroutines [50].

Swift 5.5 and onwards adheres to structured concurrency by providing the `Task` and `TaskGroup` types, along with rules that govern their lifetimes and scope. The two types are closely related, but will be explained separately in the two following subsections.

### Asynchronous contexts with `Tasks`

All `async` functions in Swift concurrency execute in some task[4], and whenever a function is suspended (at an `await`), it is technically the task running that function that is suspended. In this regard, a task is to an asynchronous function what a thread is to a synchronous function [12]. As mentioned in the previous subsection, asynchronous functions cannot be called from a synchronous context. The `Task` type provides an asynchronous context in which to invoke asynchronous functions. This context is represented as a closure passed to the initializer of `Task`, as exemplified in Listing 2.10. Tasks created by passing a closure to the `Task` initializer, as in Listing 2.10, are called *unstructured tasks*.

In the Swift concurrency model, tasks are never created implicitly; they must be created explicitly [51], as in Listing 2.10. If a task is not explicitly created when invoking an asynchronous function, two things can happen. Firstly, if the invocation occurs in a synchronous context, the code will not compile (with the error message seen at lines 7-8 and 10-11 in Listing 2.10). Otherwise, if the invocation occurs in an asynchronous context, as on line 18, the task that will run the invoked asynchronous function will be the same task as the task that was running the original context, where the invocation happened.

[4] It is unfortunate that the term *task* is used both to refer to the units of work dispatched to queues in GCD, and to `Tasks` in Swift concurrency. The distinction should be clear from usage, but will in some places be explicitly clarified by stylizing the latter in monospaced font.

```
func foo() async -> Int {
  return 42
}

// (Global, synchronous context here)

foo() // error: 'async' call in a function that
      // does not support concurrency

await foo() // error: 'async' call in a function that
            // does not support concurrency

Task {
    foo() // error: expression is 'async' but is not
          // marked with 'await'
          // note: call is 'async'

    await foo() // OK
}

// Code here will execute concurrently with
// the code inside the Task above
// (Or, in other words, creating an unstructured
// task does not block control flow)
```

Listing 2.10: Using `Task { ... }` to create unstructured tasks, providing an asynchronous context for invoking asynchronous functions.

Put concretely, if the asynchronous function `foo()` of Listing 2.10 invoked another asynchronous function, `bar()`, it would still be the task created at line 13 that ran this function (assuming `foo()` did not invoke `bar()` in a new, explicitly created task). This also means that whenever some asynchronous function invocation suspends at an `await`, this suspension will propagate up the "asynchronous invocation chain" to the place where the original task was created.

Note that creating an unstructured task does not block local control flow, unless the result value of the task is explicitly awaited. The result value of a task can only be awaited if the task is created within an already asynchronous context, as exemplified by Listing 2.11. When awaiting the result value of a task, `await` not only marks a potential suspension point within a task, but also marks a dependency between tasks: the original task will not be resumed until the task it awaits has finished.

Finally, unstructured tasks are, as their name implies, not structured, which means there are no guarantees made as to their lifetime and scope (for example, the task created at line 14 of Listing 2.11 may still be running long after the task created at line 10 has finished). For such guarantees, tasks are grouped together as parent and child tasks, by using the `TaskGroup` type.

### Task hierarchies with `TaskGroup`

The main motivation of enforcing a task hierarchy is to model dependencies between units of work. For example, there might be some work that depends on a number of subtasks before it can finish. The subtasks can be performed concurrently, and when they are done, the original work can complete. To model such a hierarchy of tasks, a `TaskGroup` can be used.

As an example, consider the code in Listing 2.12. The (asynchronous) function `foo()` calls some asynchronous functions (`bar1()`, etc.) and then combines the results in an array, to create a `Foo`, which is then returned to the caller.

Note however, that even though the `bar()` functions are asynchronous, they are called sequentially, meaning that the invocation of `bar3()` will not happen until `bar2()` has returned, and `bar2()` will not be invoked until `bar1()` has returned. Since the three function calls are independent of each other, they could be called concurrently. Listing 2.13 shows how this can be accomplished with a `TaskGroup`. The `withTaskGroup()` call provides a scope (lines 2-18) within which a `TaskGroup`, denoted 'group' in the listing, is available. Invoking `addTask()` on this group

```
func foo() async -> Int { return 42 }

// Global, synchronous context here

await Task {
    await foo()
}.value // error: 'async' property access in a function that
        // does not support concurrency

Task {
    await foo() // Control flow is blocked here
                // until foo() returns

    Task {
        await foo()
    }

    // Code here will execute concurrently with
    // the code inside the Task (line 14) above

    await Task {
        await foo()
    }.value

    // Code here will execute only when
    // the code inside the Task above
    // (line 21) has finished running
}

// Code here will execute concurrently with
// the code inside the Task (line 10) above
```

Listing 2.11: Awaiting a `Task { ... }` to block local control flow until it has completed.

```
func foo() async -> Foo {
  let res1 = await bar1()
  let res2 = await bar2()
  let res3 = await bar3()

  return Foo([res1, res2, res3])
}
```

Listing 2.12: Sequential asynchronous calls

creates a *child task* and associates it with the *parent task* (the task running the asynchronous `foo()` function).

Unlike the `DispatchGroup` of GCD, which can only wait for child tasks to complete, but not manage their return values, a `TaskGroup` *can* manage the return values of child tasks. In Listing 2.13, this is reflected at line 2, where `Bar` is passed as the result type of the group, and at lines 14-16, which aggregate the results of the child tasks into the `tmp` array (these results arrive in order of child task completion, which is not necessarily the same order in which they were started). The `for await` loop works like an ordinary `await`, so that the task is suspended (freeing up the thread to do other work) until the next result arrives. Results from a `TaskGroup` can also be aggregated in various other ways, for example using functions like `map`, `filter`, and `reduce`.

Note that if the results were not explicitly collected, as they are in the loop at lines 14-16, the runtime would still ensure that the rules of structured concurrency hold, i.e. that all child tasks have run to completion before the scope exits at line 18 (an example of this can be seen in Listing 3.2 in Chapter 3).

Also note that child tasks created within a `TaskGroup` are different from unstructured tasks. In the first case, the lifetime of child tasks are scoped by the lifetime of their parent (such that no child task can outlive its parent), while in the latter case, there are no guarantees made as to which tasks outlive which.

### Async let

Creating a `TaskGroup` for managing child tasks that can run concurrently works well when the child tasks' result types are all the same (as `Bar` above), or when there is a dynamic amount of child tasks to create. However, when there is a fixed, statically known (i.e. when writing the code) amount of child tasks, and especially when the results of the child tasks are of different types, the `async let` keywords can be used. In a regular `let` statement, the declared identifier becomes a constant, meaning that it cannot be assigned to later (like when using the `const` keyword in C or Java for example). Initializing a constant with the `async let` syntax creates a child task of the current task [52]. The child task runs concurrently with the parent task, and evaluates the initializer (i.e. the expression to the right of the = sign). When the parent task needs the result, it will `await` it, which will block control flow (freeing up the thread) until the result is available.

Semantically, using an `async let` binding is similar to creating a

```
func foo() async -> Foo {
    let res: [Bar] = await withTaskGroup(of: Bar.self) { group in
        group.addTask {
            await bar1()
        }
        group.addTask {
            await bar2()
        }
        group.addTask {
            await bar3()
        }

        var tmp: [Bar] = []
        for await res in group {
            tmp.append(res)
        }
        return tmp
    }
    return Foo(res)
}
```

Listing 2.13: Structured concurrency with `TaskGroup`

`TaskGroup` with only one child task. In both cases, you spawn a child task that does some work, and then await the result when you need it (and if you do not explicitly await the results, the child task will be implicitly awaited when you leave the scope where it was created). However, according to the `async let` language proposal, the language implementation of `async let` bindings allows for some optimizations that are not possible with `TaskGroups` [52]. Listing 2.14 shows how `async let` can be used to simplify the code of Listing 2.13.

```
func foo() async -> Foo {
    async let res1 = bar1()
    async let res2 = bar2()
    async let res3 = bar3()

    return await Foo([res1, res2, res3])
}
```

Listing 2.14: Structured concurrency with `async let` syntax

Instead of awaiting the results of each call before moving on to the next call (as in Listing 2.12), three child tasks will be spawned that run concurrently with the parent task running the asynchronous `foo()` function, and all three results can be awaited at the same time on line 6. The *Fibonacci*

benchmark in Listing 3.4 in Chapter 3 contains an example of `async let` usage.

### 2.4.3 Actors

To protect shared state from being concurrently modified by several tasks at the same time, Swift 5.5 introduces the *actor* type. An actor is similar to a class, but is automatically concurrency-safe, because it provides mutual exclusion, ensuring that only one task at a time can modify its state [13]. This is called *actor isolation*, and all instance properties and instance methods of an actor type are actor-isolated by default (class properties and methods, defined with the `static` keyword, are not actor-isolated).

Any access to actor-isolated declarations from *outside* of the actor context is called a *cross-actor reference*, and is only allowed in two circumstances. If the reference is to an instance property of constant value, it is allowed and there is no need to `await` the reference. Otherwise, the reference must occur as an invocation to an asynchronous function, i.e. by using `await` (in other words, this is a potential suspension point). This means that any instance methods of an actor, whether they be declared as `async` or not, must be invoked with `await` when the invocation happens outside the actor (actor-isolated instance methods *within* an actor can call other actor-isolated instance methods on that same actor, since this is not a cross-actor reference). Listing 2.15 illustrates how these rules work.

Note the use of `Task{ ... }` to be able to invoke asynchronous functions, as described in the previous subsection.

#### The actor model

The way Swift handles data synchronization via `actor`s is based on the *actor model* [53], the most popular implementation of which can be found in the *Erlang* programming language [54]. Erlang uses the concept of a *process*[5] as a concurrency primitive which protects its internal state, and only allows for modification of that state by processing messages sent to the process. The process will only handle one message at a time, which prevents data races on the internal state of an Erlang process.

In Swift concurrency, an actor instance is similar to having a class where changes to its state are processed via a serial dispatch queue in GCD. The language proposal for actors does state, however, that "the actual

[5] Note: Not to be confused with operating system processes.

```
actor Bank {
    var x = 0
    let y = 42

    func inc() {
        x += 1
    }
}

var b = Bank()

print(b.x) // error: actor-isolated property 'x' can not
           // be referenced from a non-isolated context
print(b.y) // OK, 42

b.inc() // error: actor-isolated instance method 'inc()' can not
        // be referenced from a non-isolated context
        // note: calls to instance method 'inc()' from outside
        // of its actor context are implicitly asynchronous

Task {
    b.inc() // error: expression is 'async' but is not
            // marked with 'await'

    await b.inc() // OK

    print(b.x) // error: expression is 'async' but is not
               // marked with 'await'
               // note: property access is 'async'

    await print(b.x) //OK, 1
}
```

Listing 2.15: Declaration of, and interaction with, an `actor` type

implementation in the actor runtime uses a lighter-weight implementation that takes advantage of Swift's async functions" [13].

### 2.4.4 Summary of differences between the GCD concurrency model, and Swift concurrency

Since there are quite a few different concurrency constructs to keep track of in both models, this section provides a brief summary of the similarities and differences between the two models. The main difference lies in the distinction between blocking a *thread*, and blocking *control flow*. Both models provide a way to spawn concurrent units of work both asynchronously (i.e. without blocking control flow), and synchronously (i.e. blocking control flow). The difference is that in the GCD model, blocking control flow also blocks the thread, whereas the Swift concurrency model allows for blocking control flow without blocking the thread. Table 2.1 summarizes the differences between the two concurrency models in this regard.

Table 2.1: Summary of the distinction in the two concurrency models between blocking control flow and blocking a thread.

| | **Blocks control flow** | **Blocks the thread** |
|---|---|---|
| **Swift concurrency** | - | - |
| `await` | Yes | No |
| `async let` | Not until result is awaited | No |
| `Task { ... }` | No (unless `await`ed) | No |
| **GCD** | - | - |
| `.sync { ... }` | Yes | Yes |
| `.async { ... }` | No | No |

Also worth noting is the distinction between asynchronous functions in the two models. In GCD, an asynchronous function is a function that might return before all its work is done (because some, or all, work is submitted to a dispatch queue). They optionally take completion handlers to call when the asynchronous work is done. It is impossible to see if a GCD-function is synchronous by looking at its function signature. In Swift concurrency, an asynchronous function is a function marked with the `async` keyword. An `async` function in Swift concurrency provides an asynchronous context and can yield control of the thread to the runtime system.

Finally, Table 2.2 summarizes a few corresponding elements of the two models.

Table 2.2: Summary of corresponding elements of the two models.

| | **GCD** | **Swift concurrency** |
|---|---|---|
| **Managing groups of tasks** | `DispatchGroup` | `TaskGroup` |
| **Synchronize access to shared resource** | Class with private serial dispatch queue | Define type as `actor` |

## 2.5 Benchmarking

A *benchmark* is a computer program designed specifically to measure the performance of a computer system, or compare the performance between several computer systems. Among other things, this is useful when deciding which computer system to select to run some application(s) of interest, where performance needs to be taken into account. While the best approach would likely be to run the actual application programs of interest on the computer system(s) and measure their performance, this might not be possible, e.g. if these applications have not yet been developed or because the time or cost to develop them for each individual system would be too great. In these cases, the performance of benchmark programs can be used to estimate what the performance of the real programs might look like [55].

Different kinds of benchmark programs with different purposes have historically been proposed. *Synthetic* benchmarks are benchmarks that try to emulate the same "instruction mix" as the real application programs (e.g. frequent memory fetches, string manipulation, large-scale vector operations, etc.), without really doing any "useful" work. One problem associated with synthetic benchmarks is that they fail to capture the patterns of instruction orderings and memory locality of the real application programs, both of which may affect performance considerably. For this reason, they may perform differently depending on e.g. compiler optimizations [55].

*Microbenchmarks* are benchmarks that are designed to test a small and specific part of a system [55], such as how efficiently two components can communicate, or the latency of some class of API calls, and so on.

*Application benchmarks* are benchmarks that actually perform some real work, often developed with certain runtime characteristics in mind. These benchmarks can typically be more accurate in emulating the behaviour of the real applications of interest. They can still be limited, especially in their

tendency to use small input data samples to ensure that running them takes a reasonable amount of time, but they are still claimed to be one of the most realistic types of benchmarks [55].

In short, benchmarks are surrogate programs used to model the behaviour of real programs, and one could say that different types of benchmarks differ mostly in their extent of how much of the real programs that they try to emulate, which impacts how realistic they are. Choosing benchmarks with care is important in order to ensure that they are representative of the programs they mimic. Moreover, as is always the case when using one thing to model something else, one must be cautious with how the results are interpreted.

Once benchmarks have been developed, the next question becomes what to measure, and how to analyse these measurements. This is discussed in the following two subsections.

### 2.5.1 Performance metrics

A *performance metric* is a value of interest that can be used to summarize some performance aspect of a computer system. This could be e.g. a count of how many memory allocations occur when running a benchmark, or the total execution time, or some other value. In "Measuring computer performance – A practitioner's guide", D. Lilja identifies six properties of good performance metrics, a few of which are:

- *linearity* (the metric should scale proportionately with performance; if the performance metric of one system is twice as good as the value of another system, the first system should outperform the second one by a factor of two)
- *reliability* (if a performance metric of one system (A) indicates that it should outperform some other system (B), then system A should always outperform system B)
- *ease of measurement* (if a performance metric is hard to measure, the risk increases of it being measured incorrectly, to the detriment of the subsequent performance analysis)

Lilja presents *execution time* as a performance metric that fulfills all six identified properties [55]. Further introduced is the concept of *means-* vs *ends-based* metrics. Consider, for example, measuring how many instructions can be performed in a certain fixed time interval. This may not be the best way

to capture the potential performance of a computer system, since different kinds of instructions may perform different amounts of work (consider *reduced instruction set computer architectures* (RISC) versus *complex instruction set computer architectures* (CISC), for example). Measuring "instructions per second" would be classified as a means-based metric, since it measures how much effort was exerted by the system, but not the usefulness of that effort, i.e. by how much it contributed to reaching an end result. Execution time is classified as an ends-based metric, because by definition it measures exactly how efficient a computer system was at reaching a goal (i.e. running the benchmark program to completion). Any unnecessary work performed, no matter how efficiently it was executed, does not improve the value of the performance metric (instead, it makes it worse).

One thing to consider when measuring execution time is whether to measure *wall clock time*, or *CPU time*. The first measures the total time to execute a benchmark, even when the CPU is not actively running the benchmark (such as when other programs time-share the CPU, or when system operations are performed, etc.). It is analogous to starting a stopwatch when starting the run of a benchmark, and stopping the stopwatch when the benchmark has finished. This method is suitable to capture the amount of time the user of a computer system will actually have to wait. When measuring *CPU time*, time is only measured when the benchmark program is actually running on the CPU [55].

### 2.5.2 Statistical analysis

After the measurement results have been gathered, the next step is to analyse the data to be able to draw conclusions from the experiments. When measuring execution time, there are many non-deterministic system effects that affect the measurements, such as cache misses, background system processes, network chatter, and so on. To mitigate these effects, it is common practice to run the same experiments several times, and summarize the measurement data.

In statistical terminology, the outcome of each experiment is represented by the *random variable* $\boldsymbol{X}$ taking a value $x$, the value measured as the outcome of the experiment. The random variable $\boldsymbol{X}$ has a *probability distribution* which captures the notion of "how likely it is to observe a certain value". The *expected value* of the random variable, $\mu$, represents what the value of the variable is expected to be, if it was sampled infinitely many times, and the results were averaged.

### Different kinds of means

To summarize a set of measured values, the *mean* of the values is commonly calculated. Depending on the kinds of measured values, different kinds of means are suitable. Arguably the most common mean is the *artithmetic mean*, which is the sum of all the measured values, divided by the amount of values:

$$\overline{x}_A = \frac{1}{n}\sum_{i=1}^{n} x_i$$

This mean is what typically should be used when the sum of the measured values would, in itself, be a meaningful and interesting value [55]. For example, the total execution time of all iterations of a benchmark is a meaningful value, and thus the arithmetic mean is suitable to use for summarizing these measurements.

The *harmonic mean* should be used when the measured values represent rates of something (memory allocations per second, for example). The harmonic mean is calculated similarly to the arithmetic mean, but with taking the inverse of each measurement value in the summation, and dividing the number of measured values by that sum:

$$\overline{x}_H = \frac{n}{\sum_{i=1}^{n} 1/x_i}$$

Note that the example of memory allocations per second is a good example of when the sum of the measured values is not a meaningful value in itself.

### Variance and standard deviation

Summarizing all measurements to arrive at a single value discards information about the distribution of the original data. For example, the data points may vary a lot, or they may be very concentrated around a certain value. The measure of this is called the *population variance* of the data, $\sigma^2$, and is calculated as:

$$\sigma^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - \mu)^2$$

The square root of the variance, called the *population standard deviation*, denoted $\sigma$, is more often used to report the spread of the data:

$$\sigma = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \mu)^2}$$

When the true value of $\mu$ is unknown, one must instead approximate it by using the *sample mean*, $\bar{x}$. This sample mean can then be used to calculate the *sample variation*:

$$s^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2$$

and *sample standard deviation*:

$$s = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2}$$

### Confidence intervals

Since the actual, true (theoretical) mean of the data, $\mu$, is often unknown, it is approximated by $\bar{x}$, the sample mean of the data. A *confidence interval* of confidence level *C* is then usually formed around $\bar{x}$. A confidence interval is a numerical interval centered around $\bar{x}$ that, along with its confidence level, *C*, quantifies how "certain" we are that the interval covers $\mu$. If we were to repeat our experiments infinitely many times, each time getting a new $\bar{x}$ and thus a new confidence interval, then C% of those intervals would cover the true value of $\mu$. Confidence levels of 95% or 99% are commonly used.

The formula for calculating a confidence interval differs depending on a few things, e.g. whether or not the underlying probability distribution of the data points is normally distributed, or whether or not the true variance, $\sigma^2$, is known or unknown.

For the case where the underlying probability distribution is unknown, which is what will be relevant for this study, the *Central Limit Theorem* (CLT) can be used. This theorem states that when the sample size, *n*, increases, the distribution of

$$\frac{\bar{\mathbf{x}} - \mu}{\sigma/\sqrt{n}}$$

approaches the *standard normal distribution* [56], i.e. a normal distribution with a mean of zero and standard deviation of one, $N(0, 1)$.

What this means in practice is that for "sufficiently large" sample sizes (n > 30 is a common threshold, though this will be discussed further in Chapter 5), it does not matter what the underlying probability distribution is. An *approximate* confidence interval for the unknown parameter $\mu$ is then given by

$$(\bar{x} - z_{\alpha/2} \cdot s/\sqrt{n}, \quad \bar{x} + z_{\alpha/2} \cdot s/\sqrt{n})$$

where $s$ is the sample standard deviation, and $z_{\alpha/2}$ is a value defined in a z table [57]. For a confidence level of 95%, the value of $z_{\alpha/2}$ is 1.96.

### Hypothesis testing

When comparing two different sets of data points, we may be interested to know whether the difference between their sample means, $\bar{x}_1$ and $\bar{x}_2$ (approximating $\mu_1$ and $\mu_2$, respectively), is *statistically significant*, i.e. different enough to suspect that the measured quantities are, in reality, different.

To investigate this, it is customary to define a *null hypothesis*, $H_0$, assuming there is no difference, i.e. $\mu_1 - \mu_2 = 0$, and an alternative hypothesis, $H_1$, assuming there is a difference, i.e. $\mu_1 - \mu_2 \neq 0$. To test the hypothesis, an *approximate* confidence interval of the difference between the means can be formed:

$$((\bar{x}_1 - \bar{x}_2) - z_{\alpha/2} \cdot S, \quad (\bar{x}_1 - \bar{x}_2) + z_{\alpha/2} \cdot S)$$

where

$$S = \sqrt{s_1^2/n_1 + s_2^2/n_2}$$

and $s_1^2$ and $n_1$ are the sample variance and sample size of the first set of data points, and $s_2^2$ and $n_2$ are the sample variance and sample size of the second set of data points.

If 0 lies within the confidence interval, then we cannot assume that there is a statistically significant difference between the two data sets, and hence we must accept the null hypothesis, $H_0 : \mu_1 - \mu_2 = 0$. If 0 does *not* lie within the confidence interval, then we *can* conclude that there is a statistically significant difference, and so reject the null hypothesis in favour of the alternative hypothesis, $H_1 : \mu_1 - \mu_2 \neq 0$. The confidence level of the interval formed, $C$, determines how confident we are of our conclusion.

## 2.6 Related work

This section contains a brief overview of related work and how it inspired this project.

### 2.6.1 The INNCABS benchmark suite

Thoman, P. et al. [2] evaluate the performance of the task-based parallelism constructs introduced in the C++11 standard, over three of the most common C++11 library implementations. To this end, they create a benchmark suite, INNCABS, containing 14 benchmarks with three different categories of task parallelism: *recursive*, *loop-like*, and *co-dependent* (see Figure 2.1).

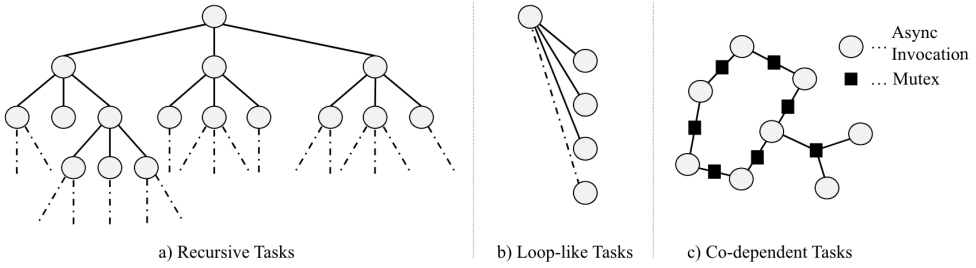



Figure 2.1: Different benchmark categories of the INNCABS benchmark suite. Figure from [2].

Recursive benchmarks are those where asynchronous tasks in turn spawn more asynchronous tasks, forming a tree-like structure. The recursive category is further divided into *balanced* and *unbalanced* recursion, based on whether each task spawns the same amount of child tasks or not (i.e. whether or not the task-tree is balanced). Loop-like benchmarks feature asynchronous tasks being spawned in a traditional loop. Co-dependent benchmarks are those where many unrelated asynchronous tasks are spawned, which cooperate via synchronization primitives like mutexes. The INNCABS benchmarks are also categorized as *coarse-grained* or *fine-grained*, depending on the amount of work done by each task.

This thesis bases the benchmark selection on the categories suggested by Thoman, P. et al., and features recursive (both balanced and unbalanced as well as fine-grained and coarse-grained) benchmarks and one loop-like benchmark. The co-dependent category is not deemed relevant to evaluate, due to the differences between the Swift concurrency model and the C++11 task-parallel model.

### 2.6.2 Performance analysis of Kotlin coroutines

Chauhan, K. et al. [58] investigate the performance of Kotlin coroutines on the Android platform. Kotlin coroutines are similar to asynchronous functions in the Swift concurrency model in many ways. Functions that can suspend are marked with the the `suspend` keyword (similar to `async` in Swift concurrency). In Kotlin, suspensions are implemented with `Continuation` objects, which consume much less memory than Java Virtual Machine (JVM) Threads (a few hundred bytes versus up to a megabyte). Kotlin coroutines also adhere to structured concurrency, like Swift concurrency tasks.

Chauhan, K. et al. identify some conditions for reliable and accurate benchmark results on the Android platform. These conditions include ensuring stable clock speeds, avoiding thermal throttling, and making sure that there is similar resource contention from background tasks in all benchmarks. Additionally, each benchmark is run in *release mode* to disable debug checks.

This thesis does not consider the clock speed of the iOS device when running the benchmarks, because the possibilities of influencing the clock speed (or even measuring it) on the platform are too limited. Thermal throttling is likewise difficult to measure. However, resource contention is limited by disabling background processes and platform utilities (such as Bluetooth, location services, etc.). Each benchmark is also run in release mode.

### 2.6.3 Integrating task parallelism and actors

S. M. Imam and V. Sarkar [59] combine the *async-finish* model (AFM) of the X10 programming language [60] with the *actor model* (AM) in what they call a "unified model". This unified model is implemented as extensions of Habanero-Java[6] and Habanero-Scala, and is very similar to the Swift concurrency model (which arguably also aims to combine these two models). For instance, the `async/finish` pair of the unified model (and the AFM) works similarly as creating a `TaskGroup` in Swift concurrency, and awaiting the spawned child tasks.

The performance of the unified model is evaluated against the JVM-based actor frameworks Jetlang, Kilim, Scala actors, and Akka (Erlang was excluded because it does not run on the JVM). Importantly, they use the Java Grande Forum *Fork-Join benchmark*, measuring the time to create actor instances that perform some small amount of work. This inspired the *SpawnManyActors* benchmark of this thesis, described in the next chapter.

[6] http://faculty.knox.edu/dbunde/teaching/hj/

# Chapter 3

# Method

To evaluate the runtime performance of the new concurrency model, six different benchmarks and an iOS application, *BenchApp*, were created. The benchmarks were all implemented in two versions: one based on the old concurrency model (the *GCD version*) and one based on the new concurrency model (the *Swift concurrency version, SC*). The app contained the benchmarks and the means to configure and run them, as well as code measuring the performance metrics. The measurement results were printed to the Xcode console[1] as the app was running the benchmarks, and this output was parsed by a Python script to generate the graphs presented in Chapter 4.

This chapter describes:

- the *research process* (how the necessary information for carrying out the project was gathered, and how the survey of related work was conducted),
- the *experimental design process* (how the performance metrics were chosen and measured, how the benchmarks were selected and implemented, how similarity between the two versions of each benchmark was verified, and the configuration setup), and
- the *data analysis process* (how the results were parsed and statistically analysed)

The code for the BenchApp application and the Python script is available online at `https://github.com/andreaskth/evaluating-swift-concurrency`.

[1] Xcode is an integrated development environment (IDE) from Apple, designed to develop applications for macOS, iOS, and other Apple platforms.

## 3.1 Research Process

The study began by researching the Swift programming language, mainly in *"The Swift Programming Language"* book [61]. Then, the new concurrency model was studied by watching *Apple Worldwide Developers Conference (WWDC)* videos [62, 51, 63, 10] and reading Swift evolution proposals [3, 12, 13]. The main source of information for the GCD concurrency model was the "Concurrency Programming Guide" in the Apple documentation archive [8].

Academical background for the concurrency models was then reviewed by searching Google Scholar (`https://scholar.google.com/`) for terms like 'structured concurrency', 'continuation passing style concurrency', 'coroutines', etc.

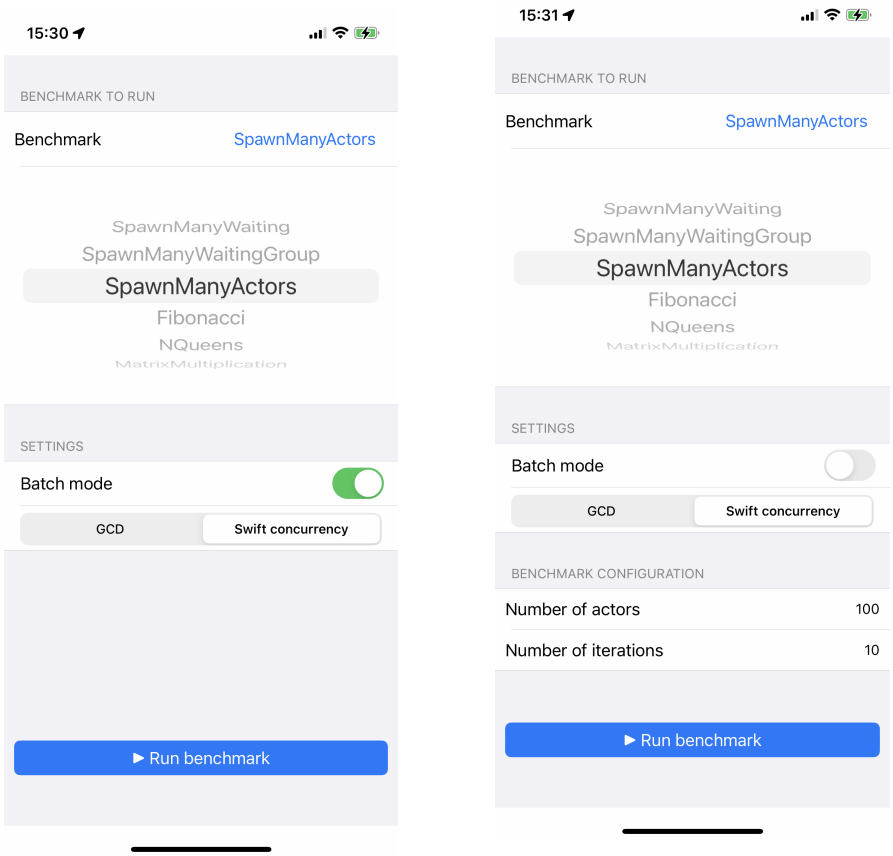
Lastly, for the survey of related work, approximately 30 potential papers were examined. They were found by searching Google Scholar with the search terms 'concurrency benchmark task-based', '"async/await" benchmark', and 'asynchronous concurrency benchmark'. For each search term, the top 10 results were briefly examined, and each paper was studied in more detail when it seemed relevant. Common benchmarks used in these studies were identified. The studies surveyed also lead to other potential papers, which were also included in the survey when deemed relevant.

## 3.2 Experimental design

The chosen methodology for evaluating the differences in execution time and memory consumption in the new concurrency model versus the old is to implement benchmarks that exercise different components of the models, and then evaluate the performance of these benchmarks based on the identified performance metrics (further discussed in Section 3.2.1).

To this end, an iOS application, *BenchApp*, was developed, with the help of the host company. The app consists of a single view, where a benchmark can be selected and then configured and started in that same view. Figure 3.1 shows what this looks like.

It is possible to either manually specify settings for the benchmark to run (*manual mode*), or to select *batch mode*, which will run the benchmark with programmatically predefined settings (these settings are further discussed in Section 3.2.3).




(a) Batch mode runs the benchmark with programmatically predefined settings.

(b) Manual (single run) mode allows benchmark settings to be configured in the app.

Figure 3.1: User interface of *BenchApp*, with the *SpawnManyActors* benchmark and *Swift concurrency* version selected.

### 3.2.1 Performance metrics

The project measures two performance metrics: execution time, and memory consumption profile. As an ends-based metric [55], execution time is well-suited to evaluate computer performance, and satisfies all the properties of a good performance metric (see Section 2.5.1). Memory consumption is interesting to measure, because apart from execution time, it constitutes the most commonly regarded finite resource that developers must consider when constructing computer programs. Additionally, the problem of thread explosion brought up in Section 1.2.1 is interesting to analyse from the perspective of memory consumption.

While it would have been interesting to also include metrics such as energy consumption or number of threads created, measuring these was found to be too demanding within the scope of the project, and were omitted to allow more detail for the two chosen metrics.

For measuring execution time, the `systemUptime` property of `ProcessInfo.processInfo` was used. This API is the recommended way to measure time in Swift, and in turn invokes `mach_absolute_time()` [64], which increments monotonically [65] (meaning that time never flows backwards) and with nanosecond precision [66]. This API measures *wall clock* time (also described in Section 2.5.1), which means that time is measured even when the program's instructions are not actively executed by the CPU (such as when other programs time-share the CPU, or when system operations are performed, or while the CPU is idle waiting for I/O such as network, etc.) [55]. One drawback with using wall clock time is that extra noise may be introduced due to underlying system effects, which would not have been present when measuring *CPU time* (which only ticks when our application is running). An advantage is that it more closely reflects the total time it takes to perform operations on an actual system.

To measure memory consumption, the `task_info` API was used [67]. It provides information about the current memory footprint of an application via the `phys_footprint` property.

Section 3.2.3 contains more details about how execution time and memory consumption were sampled during the benchmark runs.

### 3.2.2 Selecting benchmarks

Based on the literature review of related work, common types of benchmarks with different characteristics were identified, featuring fine- and coarse-grained recursion (both balanced and unbalanced), and loop-like parallelism, as established for the INNCABS benchmark suite [2]. These inspired the selection of benchmarks for this project. This section will explain each benchmark in more detail.

The six benchmarks can be divided into two groups, the *base benchmarks*, and the *applied benchmarks*. The three base benchmarks are microbenchmarks (as described in Section 2.5), each aimed at examining one specific component of the concurrency models, and the three applied benchmarks are application benchmarks (also described in Section 2.5) that use the concurrency models to solve some computational problem.

```swift
// GCD version
for _ in 1...n {
    let _ = DispatchQueue.global().sync() {
        Thread.sleep(forTimeInterval: 0.1)
    }
}

// ...

// Swift concurrency version
for _ in 1...n {
    let _ = await Task {
        try await Task.sleep(nanoseconds: 100_000_000)
    }.result
}
```

Listing 3.1: Main part of the *SpawnManyWaiting* benchmark

### Base benchmarks

Three benchmarks were developed to exercise the three different pillars of the new concurrency model: `async/await`, structured concurrency, and actors. The respective benchmark for each pillar is *SpawnManyWaiting*, *SpawnManyWaitingGroup*, and *SpawnManyActors*.

#### *SpawnManyWaiting*

This benchmark spawns $N$ tasks that wait for 0.1 seconds and then finish. It is inspired by Chauhan, K. et al. [58] and Wheeler, K. B. et al. [68] and is meant to measure and compare overhead of task creation and invocation of async code. In the GCD version, the tasks are spawned as blocks that are synchronously dispatched to the global dispatch queue with default quality-of-service. Each such task uses `Thread.sleep` to wait. In the Swift concurrency version, the tasks are spawned as `Tasks`, that invoke `Task.sleep` to wait. Note that in both cases, the tasks are running sequentially, i.e. at any point in time there is only one task executing. Listing 3.1 shows an excerpt of the benchmark code.

It might seem paradoxical to compare invocation of *async* code with `await` in the Swift concurrency version, with *sync* dispatching of code in the GCD version. This is done because we want code flow at the call site to be blocked until the call has finished. In GCD, this is expressed by synchronously dispatching tasks, and thus blocking the thread, whereas in Swift concurrency it is expressed by `await` at the call site, (potentially) suspending the task

but freeing up the thread. It is important to keep this distinction in mind. This duality between GCD and Swift concurrency, and its impact on the benchmark implementations, will be further discussed in Chapter 5.

Another point to note regarding the Swift concurrency version is the explicit creation of a `Task` (line 12) in each loop iteration, as opposed to just running the `Task.sleep` call immediately. As explained in Section 2.4.2, this is because tasks have to be explicitly created; if you do not create a `Task` when calling an asynchronous function, the called function will run as part of the same `Task` as the one running the current function [51]. This is often the desired behaviour, but is unsuitable in this benchmark, since we seek to measure overhead of task creation.

#### *SpawnManyWaitingGroup*

This benchmark is meant to measure and compare the overhead of creating and managing groups of tasks. It is similar to *SpawnManyWaiting*, with the difference that the *N* tasks instead execute concurrently, as part of the same group of tasks. In the GCD version, this is achieved with the `DispatchGroup` class [69]. As mentioned in Section 2.3.1, when tasks are submitted to a dispatch queue, a reference to a dispatch group can be passed as a parameter, and invoking `wait()` on the group will block execution (blocking the thread) until all tasks associated with the group have finished.

In the Swift concurrency version, the same behaviour is achieved with the `TaskGroup` structure [70], which works similarly to how `DispatchGroup` works, with two notable differences. Firstly, waiting for the child tasks in the group to finish is implicit (and does not block the thread, but only the parent task). This is why there is no `group.wait()` call in the Swift concurrency version. Secondly, the `TaskGroup` provides an API for collecting the results of the submitted child tasks. This will be relevant in the applied benchmarks. In this benchmark however, the results of the child tasks are irrelevant, which is why the return type is set to `Void.self`. Listing 3.2 shows an excerpt of the benchmark code for the *SpawnManyWaitingGroup* benchmark.

Note that in the GCD version, we are now dispatching tasks asynchronously to the dispatch queue (and not synchronously, as in the previous benchmark). This is because now the intended behaviour is for *N* number of tasks to execute concurrently, and for an iteration of the benchmark to finish when all the tasks of the group have finished.

```swift
// GCD version
let group = DispatchGroup()
for _ in 1...n {
    DispatchQueue.global().async(group: group) {
        Thread.sleep(forTimeInterval: 0.1)
    }
}
group.wait()

// ...

// Swift concurrency version
await withThrowingTaskGroup(of: Void.self) { group in
    for _ in 1...n {
        group.addTask {
            try await Task.sleep(nanoseconds: 100_000_000)
        }
    }
}
```

Listing 3.2: Main part of the *SpawnManyWaitingGroup* benchmark

### *SpawnManyActors*

This benchmark spawns $N$ "actor instances", i.e. instances of a class that synchronizes access to its internal state in a concurrency-safe manner. For each spawned actor instance, a method on that instance is invoked, and the instance is then discarded. The benchmark is meant to measure and compare the overhead of creating actor instances, and is inspired by the Swift concurrency roadmap [11], and the Java Grande Forum *Fork-Join benchmark* [71], as described by S. M. Imam and V. Sarkar [59].

In the GCD version, an actor is represented by a class with a private serial dispatch queue, because this was typically how classes ensured sequential access to shared data prior to the introduction of Swift concurrency [11, 33]. In the Swift concurrency version, an actor is represented by a type defined with the `actor` keyword introduced in Swift version 5.5. As mentioned in Chapter 2, actor types automatically protect their mutable state and sequentialize access to it. Listing 3.3 shows an excerpt of the benchmark code for the *SpawnManyActors* benchmark.

Note that this benchmark does *not* measure the scenario of many concurrent accesses to the same actor. This is instead measured in the *MatrixMultiplication* benchmark.

```
class GCDActor {

    private var counter = 0
    private let counterQueue = DispatchQueue(label: "GCDActor")

    func inc() {
        counterQueue.sync {
            self.counter += 1
        }
    }
}

actor Actor {

    private var counter = 0

    func inc() {
        self.counter += 1
    }
}

// ...

// GCD version
for _ in 1...n {
    GCDActor().inc()
}

// ...

// Swift concurrency version
for _ in 1...n {
    await (Actor()).inc()
}
```

Listing 3.3: Main part of the *SpawnManyActors* benchmark

### Applied benchmarks

Apart from the three base benchmarks, three additional benchmarks were implemented, to represent more realistic usage of the concurrency primitives. They were selected to cover different kinds of concurrency categories, inspired by the categories from the INNCABS benchmark suite [2]. The three categories (as described in Section 2.6) are *recursive and fine-grained*, *recursive and coarse-grained*, and *loop-based*. The three respective benchmarks are *Fibonacci*, *NQueens*, and *MatrixMultiplication*.

#### *Fibonacci*

This benchmark contains a naive, non-memoized, recursive implementation of `fib()`, a function that calculates the *n*th number of the Fibonacci sequence. It is a fine-grained and balanced benchmark, with `fib()` recursively calling itself until the base case of `n < 2` is reached. The granularity of the benchmark could be varied by adding a *threshold value*, and switching to a sequential Fibonacci calculation for calls with `n < threshold_value`, but due to time constraints, this was not implemented.

In the GCD version, the function takes a completion handler as one of its arguments, which it invokes with the result of calculations performed asynchronously on the global concurrent dispatch queue with default quality-of-service. In the Swift concurrency version, the `async let` keywords are used to concurrently calculate `fib(n-1)` and `fib(n-2)`, and the addition of these two expressions is performed when both results are ready. As mentioned in Section 2.4.2, `async let` can, at least semantically, be thought of as creating a `TaskGroup` where the `async let` initializers (i.e. the expressions to the right of the = sign) run as child tasks, and the results are awaited in expressions that use them. Listing 3.4 shows an excerpt of the benchmark code for the *Fibonacci* benchmark.

Note the `@escaping` annotation (explained in Section 2.1.1) in the GCD version. This is needed because the closure argument, `ch`, that is passed to the function is only invoked once the block submitted to the dispatch queue is actually scheduled to run, which might be after the `fib` function has returned.

#### *NQueens*

This benchmark is a recursive implementation that solves the *NQueens* problem, which is to find all ways to place *N* queens on an *NxN* chess board such that no queen can attack any other queen. The implementation is based on the C++ implementation in the INNCABS benchmark suite [72], and is slightly

```swift
// GCD version
private func fib(_ n: Int, ch: @escaping (Int) -> Void) {
    if (n < 2) {
        ch(n)
        return
    }
    DispatchQueue.global().async {
        self.fib(n-1) { res in
            self.fib(n-2) { res2 in
                ch(res+res2)
            }
        }
    }
}

// ...

// Swift concurrency version
private func fib(_ n: Int) async -> Int {
    if (n < 2) {
        return n
    }
    async let res  = fib(n-1)
    async let res2 = fib(n-2)
    return await res+res2
}
```

Listing 3.4: Main part of the *Fibonacci* benchmark

```swift
// GCD version
func solutionsGCD(_ n: Int, _ col: Int = 0,
                  _ history: [Int] = [Int]()) -> Int {
    if (col == n) {
        return 1 // If we reach this, it means all n queens were
                 // placed correctly, hence a solution was found
    } else {
        let group = DispatchGroup()

        var res = 0
        let resUpdateQueue = DispatchQueue(label: "resUpdateQueue")

        for row in 0..<n {
            let newHistory = history + [row]
            if (valid(n, col, newHistory)) {
                DispatchQueue.global().async(group: group) {
                    let numberOfSolutions =
                        self.solutionsGCD(n, col + 1, newHistory)
                    resUpdateQueue.async {
                        res += numberOfSolutions
                    }
                }
            }
        }

        group.wait()
        resUpdateQueue.sync {}
        return res
    }
}
```

Listing 3.5: Main part of the GCD version of the *NQueens* benchmark

more coarse-grained than the *Fibonacci* benchmark. Because new tasks are only spawned when a queen placement is calculated to be valid, the benchmark is unbalanced (i.e. some tasks spawn more child tasks than others).

Both versions follow the same algorithm. The algorithm loops through the squares of the chess board in a top-to-bottom, left-to-right fashion, tentatively placing a queen at each square and invoking a function, `valid()`, that checks if a queen placement is valid or not. If a placement is valid, an asynchronous task is dispatched that recursively invokes the *NQueens* function to calculate how many solutions that are possible based on the current placement. An excerpt of the benchmark code is shown in Listings 3.5, 3.6, and 3.7.

In the GCD version, a `DispatchGroup` is used to manage child tasks that share the same configuration of pieces. In the Swift concurrency version,

```swift
// Swift concurrency version
func solutionsAsync(_ n: Int, _ col: Int = 0,
                    _ history: [Int] = [Int]()) async -> Int {
    if (col == n) {
        return 1 // If we reach this, it means all n queens were
                 // placed correctly, hence a solution was found
    } else {
        return await withTaskGroup(of: Int.self) { group in
            for row in 0..<n {
                let newHistory = history + [row]
                if (valid(n, col, newHistory)) {
                    group.addTask {
                        await self.solutionsAsync(n, col + 1,
                                                  newHistory)
                    }
                }
            }
            return await group.reduce(0, +)
        }
    }
}
```

Listing 3.6: Main part of the Swift concurrency version of the *NQueens* benchmark

```swift
func valid(_ n: Int, _ col: Int, _ history: [Int]) -> Bool {
    if (col == 0) {
        return true // col == 0 means no queens have been placed
                    // yet, thus 'valid' is trivially true
    }
    let row = history[col]
    for prevCol in 0..<col {
        let prevRow = history[prevCol]
        if (row == prevRow) {
            return false // No two queens on the same row
        }
        if (col - prevCol == abs(row - prevRow)) {
            return false // No two queens on the same diagonal
        }
    }
    return true
}
```

Listing 3.7: The `valid()` function used in the *NQueens* benchmark

this is managed by a `TaskGroup`. Note that unlike in the *SpawnManyWaitingGroup* benchmark, where the result type of the `TaskGroup` was set to `Void.self`, here we are actually interested in the results of each child task, and so the result type is set to `Int.self`, and the results can be aggregated by using the `reduce()` function on the group.

In the GCD version, there is no API for collecting the results of the child tasks, so this must be done manually. This is done with the `res` variable, along with a private serial queue, `resUpdateQueue`, which is needed to prevent a data race on the `res` variable, since multiple asynchronous tasks may try to modify `res` at the same time.

At the end of the GCD version, an empty task is synchronously dispatched via `resUpdateQueue.sync`, in order to wait for all async modifications of `res` to finish.

#### *MatrixMultiplication*

This benchmark is a simple implementation of matrix multiplication and is meant to exhibit loop-like parallelism, where all cells of the result matrix are calculated concurrently with each other. An excerpt of the benchmark code is shown in Listings 3.8 (for the GCD version), 3.9 (for the Swift concurrency version), and 3.10 (for the actor used in the Swift concurrency version).

In both versions, the algorithm works by looping through each cell of the result matrix and spawning tasks that calculate the dot product of the relevant column and row of the input matrices. The difference is that in the GCD version, this is done with a `DispatchGroup`, where child tasks update the `res` matrix via a private serial queue, `resUpdateQueue` (like in the *NQueens* benchmark). In the Swift concurrency version, a `TaskGroup` is used, and asynchronous modifications to the result matrix are synchronized via a `Matrix` actor, shown in Listing 3.10.

In the GCD version, modifications to the `res` matrix are dispatched as *synchronous* tasks to the `resUpdateQueue`, as opposed to *asynchronously* as in the *NQueens* benchmark. This is done to ensure similar behaviour to the Swift concurrency version, where the `await res.update(...)` call will block local control flow (but not the thread). Note that in both versions, this blocking of control flow occurs within the asynchronous tasks, which means that the outside loop is unaffected, and can keep dispatching asynchronous tasks for the rest of the cells.

Note that in the Swift concurrency version, waiting for all tasks in the `TaskGroup` to finish is not done with a `group.wait()` call as in the GCD version, but instead by `await`ing the `withTaskGroup(...)` call.

```swift
// GCD version
func GCDMatrixMult(leftMatrix: [[Int]], rightMatrix: [[Int]])
    -> [[Int]] {

    let group = DispatchGroup()

    var res = [[Int]](
        repeating:
        [Int](repeating: 0, count: leftMatrix.count),
        count:
        rightMatrix[0].count)

    let resUpdateQueue = DispatchQueue(label: "resUpdateQueue")

    for i in 0..<leftMatrix.count {
        for j in 0..<rightMatrix[0].count {
            DispatchQueue.global().async(group: group) {
                let ownRow = leftMatrix[i]
                let otherColumn = rightMatrix.map { $0[j] }
                let cellRes = self.dotProduct(ownRow, otherColumn)
                resUpdateQueue.sync {
                    res[i][j] = cellRes
                }
            }
        }
    }

    group.wait()

    return res
}
```

Listing 3.8: Main part of the GCD version of the *MatrixMultiplication* benchmark

```swift
// Swift concurrency version
func asyncMatrixMult(leftMatrix: [[Int]], rightMatrix: [[Int]]) async
    -> [[Int]] {

    let res = Matrix(size: leftMatrix.count)

    await withTaskGroup(of: Void.self) { group in
        for i in 0..<leftMatrix.count {
            for j in 0..<rightMatrix[0].count {
                group.addTask() {
                    let ownRow = leftMatrix[i]
                    let otherColumn = rightMatrix.map { $0[j] }
                    let cellRes = self.dotProduct(ownRow, otherColumn)
                    await res.update(
                        row: i,
                        column: j,
                        newValue: cellRes)
                }
            }
        }
    }

    return await res.get()
}
```

Listing 3.9: Main part of the Swift concurrency version of the *MatrixMultiplication* benchmark

```swift
actor Matrix {

    private var m: [[Int]]

    init(size: Int) {
        m = [[Int]](
        repeating: [Int](repeating: 0, count: size),
        count: size)
    }

    func update(row: Int, column: Int, newValue: Int) {
        m[row][column] = newValue
    }

    func get() -> [[Int]] {
        return m
    }
}
```

Listing 3.10: The `Matrix` actor used in the *MatrixMultiplication* benchmark

Finally, unlike in the *NQueens* benchmark, there is no `resUpdateQueue.sync` at the end of the function in the GCD version. This is no longer necessary, since all tasks on the `resUpdateQueue` must be done if the `DispatchGroup.wait()` call has returned.

### 3.2.3 Gathering the data

This section describes how similarity between benchmarks was ensured, how the measurements were performed, and the hardware and software configurations of tools used.

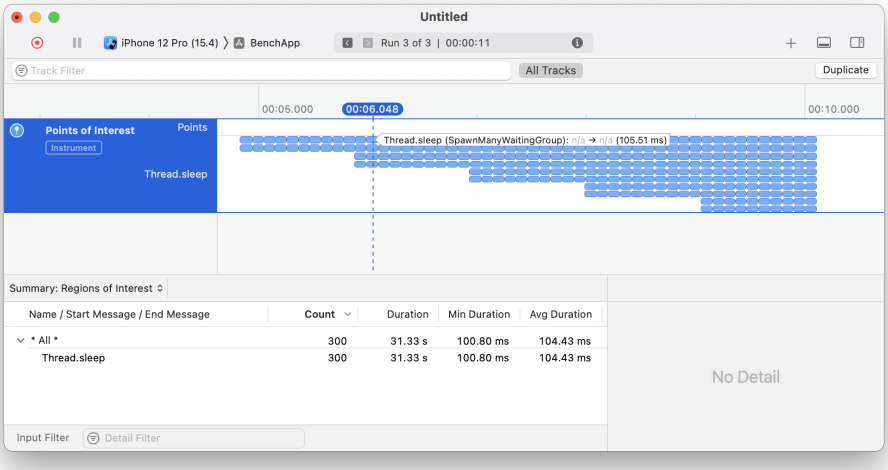
#### Ensuring similarity between benchmarks

Since queues in the GCD model have different quality-of-service that govern priority, and tasks in the Swift concurrency model also have priorities, each benchmark was instrumented with calls to `Task.currentPriority.rawValue`, which returns the priority value for the current context [73]. This was done both for the GCD and Swift concurrency versions, and the values were verified to be the same for both versions, to ensure that the comparison was fair.

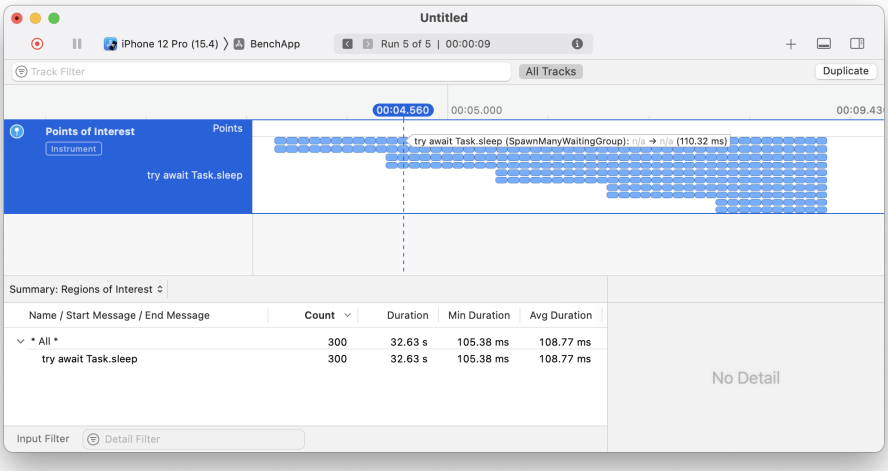
Additionally, for the base benchmarks, the `os_signpost()` API [74] was used along with the *Instruments* tool in Xcode, to ensure that "tasks" (both in the GCD sense of "blocks dispatched to dispatch queues", and the Swift concurrency sense with `Tasks`) were created and ran with equivalent patterns in both versions of each benchmark. An example of what this looks like can be seen in Figure 3.2.

#### Measuring the performance metrics

The measurements of the performance metrics were taken programmatically (i.e. by instrumenting the source code). For execution time, the measurement code ran before and after each benchmark iteration, so as to minimize interference from the measurement operations on the end result. Unfortunately, within the scope of this project, it was not possible to find a suitable method for measuring memory consumption in this manner. As mentioned in Section 3.2.1, there is an API, `task_info` [49], which provides information about the current memory footprint of an app via the `phys_footprint` property. However, sampling this value before and after a benchmark yields no information about what the memory usage looks like when the benchmark is running. Therefore, this value was repeatedly sampled as each benchmark




(a) Visualization of tasks created in the GCD version of the *SpawnManyWaitingGroup* benchmark




(b) Visualization of `Tasks` created in the Swift concurrency version of the *SpawnManyWaitingGroup* benchmark

Figure 3.2: Using Instrument's *Points of Interest* instrument to ensure similarity between benchmark versions of *SpawnManyWaitingGroup*.

was running. This was done with the `CADisplayLink` class, which allows for configuring a function to be called each time the screen is updated, which, for the hardware and software setup of this project, means 60 times per second.

A drawback with frequently sampling the memory consumption is that the `task_info` call is expensive [75], so there is inevitably some overhead that affects the measurements. The exact size of this overhead was not quantified, but should affect measurements of both versions of each benchmark the same way.

To minimize the effects of random noise on the execution time measurement data, each benchmark was run for several iterations. When running the benchmarks, a warm-up period was observed during the first few iterations (1-5), where execution times were noticeably higher. This could probably be attributed to system effects such as e.g. the contents of memory caches affecting memory fetches, which in turn affect execution time. Therefore, the benchmarks are run for 110 iterations, and the first 10 are then discarded, to ensure steady-state application performance, as suggested in other work [76]. Finally, memory consumption and execution time were measured separately, to minimize interference.

The input data for each benchmark was configured programmatically as arrays of *benchmark parameters*. Table 3.1 shows these parameters for each benchmark. For the *MatrixMultiplication* benchmark, random matrices were generated using the `drand48()` call, and the random seed was set at the start of the benchmark, using `srand48(12345)`, to ensure reproducibility.

Table 3.1: Batch mode parameters for the different benchmarks.

| **Benchmark** | **Batch mode parameters** |
|---|---|
| SpawnManyWaiting | [2, 4, 6, 8, 10] |
| SpawnManyWaitingGroup | [50, 100, 150, 200, 250, 300] |
| SpawnManyActors | [100, 200, 300, 400, 500, 600, 700, 800, 900, 1000] |
| Fibonacci | [5, 10, 15, 20, 25] |
| NQueens | [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12] |
| MatrixMultiplication | [25, 50, 75, 100] |

### Experimental setup

To reduce contention for resources by background processes on the device running the *BenchApp*, all other applications were closed. The device was set

to *airplane mode*[2], to disable utilities such as location services, Wi-Fi, and Bluetooth, and also to prevent incoming text messages or phone calls from interrupting the measurements. The device was kept charged at > 95% and actively charging, and was connected via cable (lightning to USB-C) to a MacBook Pro (M1 Pro, 2021 model) running the Xcode application, which gathered the log messages printed from the measurements. The phone was placed on a table and was not disturbed nor touched while the experiments ran, to not trigger any accelerometer or touch events.

The benchmarks were run in batch mode, with the benchmark parameters noted in Table 3.1. There are three mutually exclusive compiler flags that govern the optimization of the code. The `-Onone` flag does not perform any optimizations, the `-O` flag optimizes for speed, and the `-Osize` flag optimizes for code size. The impact of the chosen flag was measured by varying the flag when running the benchmarks. The flag was chosen in *Build Settings* in Xcode, as shown in Figure 3.3.

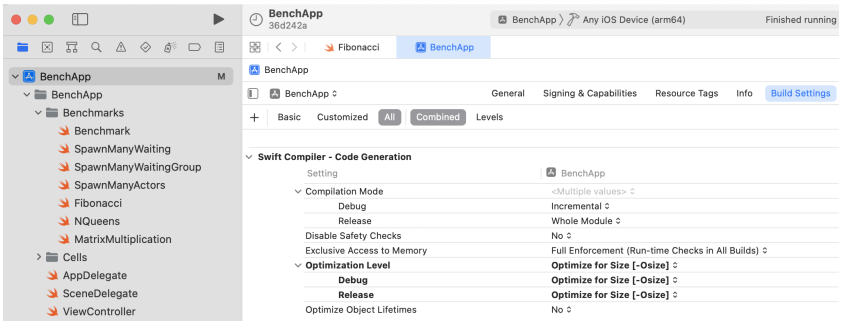



Figure 3.3: Choosing the compiler flag for optimization level in *Build Settings* in Xcode.

Each benchmark was thus run for a total of six different times across six separate runs of the app (to measure all combinations of the two concurrency versions and three different compiler optimizations enabled, `-Onone`, `-O`, and `-Osize`). The results of these runs are saved separately, and are reported in the next chapter.

### Hardware specification

The hardware specifications of the device running the *BenchApp* are listed in Table 3.2.

[2] This was done immediately *after* starting the app, as the app could not be launched otherwise.

Table 3.2: Hardware specs

| | |
|---|---|
| Model | iPhone 12 Pro |
| Model Number | MGMP3QN/A |
| Software Version | 15.4.1 (19E258) |
| RAM | 6 GB |
| Storage | 256 GB |
| Number of cores | 6 (2 high-power performance cores + 4 low-power efficiency cores) |

**Software configuration**

The software versions of the Xcode application and Instruments tool are both *13.3 (13E113)*. All benchmarks were run with the 'Run' operation in Xcode, which was set to build in 'Release' mode. This is already reflected in the code made available in the Github repo[3], but is explicitly mentioned here for clarity.

## 3.3 Data analysis

The measurement data was printed to the Xcode console as each benchmark was running. This output was saved to file, and two Python scripts (one for execution time analysis, and one for memory consumption analysis) were developed that parsed these files and generated graphs from the data. All measurement results, and the Python scripts, are available at `https://github.com/andreaskth/evaluating-swift-concurrency`.

The data analysis for execution time measurements and memory consumption measurements was performed in different ways, and will be explained separately.

### 3.3.1 Execution time

Listing 3.11 shows an abbreviated example of what the Xcode console output can look like when measuring execution time.

The Python script parses output files, using the `re` module[4] to parse the output with regular expressions, and the `matplotlib` library[5] for plotting graphs of how the execution time of a benchmark changes based on the value

[3] Specifically in the `BenchApp.xcodeproj/xcshareddata/xcschemes/BenchApp.xcscheme` file.

[4] `https://docs.python.org/3/library/re.html`

[5] `https://matplotlib.org/`

```
Pressed 'Run benchmark'
About to run 'Swift concurrency'-version of SpawnManyActors with batch mode: true
Running BenchApp.SpawnManyActors:runAsync with 100 actor(s) and 110 iteration(s).


7.812498370185494e-05
6.53750030323863e-05
6.504164775833488e-05
... 105 values omitted ...
4.7666660975664854e-05
6.208335980772972e-05

(async, average, 100 actors) Done in an average of 6.825958436820656e-05 seconds.

Running BenchApp.SpawnManyActors:runAsync with 200 actor(s) and 110 iteration(s).

0.0001287916675209999
9.099999442696571e-05
9.216670878231525e-05
... 105 values omitted ...
6.937497528269887e-05
7.85833690315485e-05

(async, average, 200 actors) Done in an average of 7.996458443813026e-05 seconds.

...
```

Listing 3.11: Abbreviated example of what execution time measurement output in Xcode might look like

of the benchmark parameter for that benchmark (e.g. number of tasks created, number of actors created, Fibonacci number to calculate, etc.). To arrive at a single data point for a certain benchmark parameter, the measurement results for all 100 iterations (after discarding the first 10) of that benchmark parameter are averaged using the arithmetic mean (see Section 2.5.2). This average value is also reported in the Xcode output.

The Python script also stores each iteration result, so that an approximate 95% confidence interval for each data point can be calculated. This approximate confidence interval is also plotted as part of the resulting graph, to show the spread of the data. The interval is calculated assuming that the mean of the data follows a normal distribution (as per the Central Limit Theorem). This assumption will be discussed further in Chapter 5.

Finally, for any given benchmark parameter, each GCD data point for that parameter is compared to each SC data point for the same parameter, by subtracting the GCD value with the SC value, and forming an approximate 95% confidence interval around this difference, as described in Section 2.5.2. This approximate interval is used to check if the difference between the two models is statistically significant.

```
Pressed 'Run benchmark'
Starting memory measurement. Max FPS is 60 FPS.
About to run 'GCD'-version of SpawnManyActors with batch mode: true
Running BenchApp.SpawnManyActors:runGCD with 100 actor(s) and 110 iteration(s).

2022-06-06 17:17:27 +0000, mem: 14.7200927734375
2022-06-06 17:17:27 +0000, mem: 14.7357177734375

(GCD, 100 actors) Done.

Running BenchApp.SpawnManyActors:runGCD with 200 actor(s) and 110 iteration(s).

2022-06-06 17:17:27 +0000, mem: 14.7357177734375
2022-06-06 17:17:27 +0000, mem: 14.7357177734375

... values for 300-900 actors omitted ...

Running BenchApp.SpawnManyActors:runGCD with 1000 actor(s) and 110 iteration(s).

2022-06-06 17:17:28 +0000, mem: 14.7357177734375
2022-06-06 17:17:28 +0000, mem: 14.7357177734375
2022-06-06 17:17:28 +0000, mem: 14.7357177734375
2022-06-06 17:17:28 +0000, mem: 14.7357177734375
2022-06-06 17:17:28 +0000, mem: 14.7357177734375
2022-06-06 17:17:28 +0000, mem: 14.7357177734375

(GCD, 1000 actors) Done.

Benchmark done
```

Listing 3.12: Abbreviated example of what memory consumption measurement output in Xcode might look like

### 3.3.2 Memory consumption

Listing 3.12 shows an abbreviated example of what the Xcode console output can look like when measuring memory consumption.

The memory consumption profile of each benchmark run, as a whole, is constructed by plotting the sampled `phys_footprint` values over time, using `matplotlib`. Unlike the execution time data points, which are measured repeatedly and averaged, the memory consumption profile is measured throughout the entire run of a benchmark, and is not averaged. The purpose is to visualize the difference in memory behaviours between the two concurrency models, both in terms of memory consumption profiles (i.e. *how* is memory consumed, is it stable or fluctuating, constant or steadily increasing, etc.) and also in terms of peak memory consumption.

# Chapter 4

# Results

This chapter contains the results of the performance measurements of the six benchmarks, with one section per benchmark. The results are analysed and discussed in the next chapter. For each benchmark, graphs of execution time based on compilation flag and benchmark parameters (tasks, actors, matrix dimensions, etc.) are included, along with graphs of the memory consumption profile as the benchmark was run (all iterations included). Each plotted line in the execution time graphs is accompanied by a shaded area of the same colour as the line, representing an approximate 95% confidence interval of that line. This gives a rough estimate of the spread of the data, but is not used to infer whether the difference between two lines is statistically significant.

To save space, memory graphs for different compilation flags are only included if the memory consumption profile differs between them; otherwise the `-Onone` version is used (all data and graphs are available at `https://github.com/andreaskth/evaluating-swift-concurrency`). The memory graphs are annotated with vertical lines to indicate the value of the benchmark parameter at different regions of the graph. Some benchmarks execute too fast to register any measurements between certain benchmark parameters, and in these cases the vertical lines and their labels overlap.

Tables containing approximate 95% confidence intervals of the difference in average execution time between the two concurrency models, i.e. $diff = GCD_{avg} - SC_{avg}$, are also included. These intervals are calculated as described in Section 2.5.2. The GCD version is always the first operand in the subtraction, so a positive number indicates a case where the SC version ran faster than the GCD version, and vice versa for negative numbers. Table cells are coloured grey when the intervals contain 0, and thus represent cases where the difference is not statistically significant at a 95% confidence level.

## 4.1 The *SpawnManyWaiting* benchmark

Figure 4.1 shows the execution time of the *SpawnManyWaiting* benchmark as a function of number of tasks created, $N$. Tables 4.1 and 4.2 report approximate 95% confidence intervals for the differences between the means of the execution times for N = 2 and N = 10 (tables for all values of N are available in Appendix A as Tables A.1-A.5).

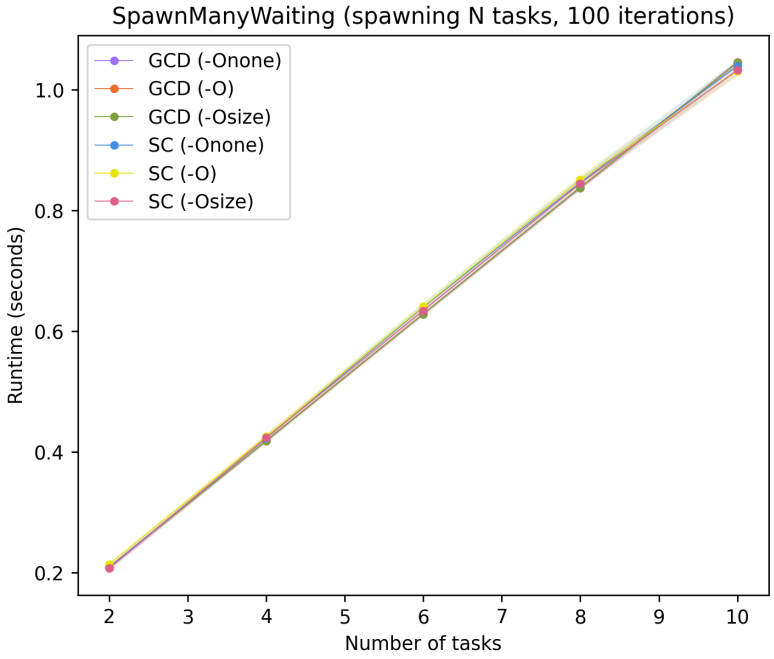
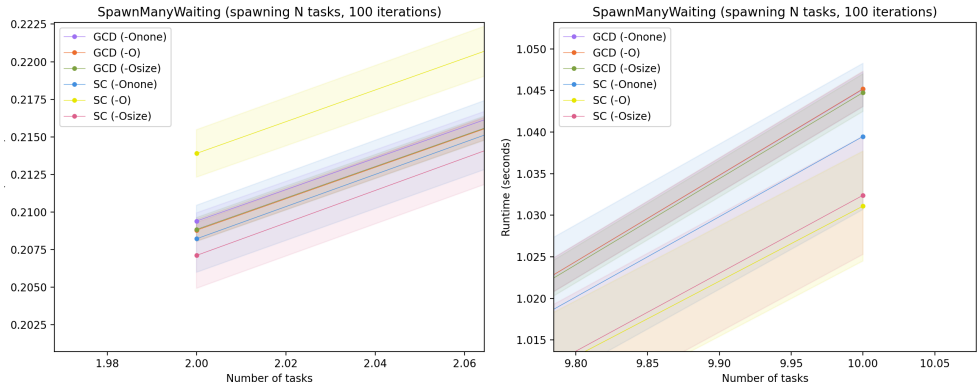



Figure 4.1: Overview of execution time for the *SpawnManyWaiting* benchmark with all benchmark parameters.

Overall, the difference between the two concurrency models is generally small, but mostly statistically significant. For N values of 4, 6, and 8, all GCD versions (`-Onone`, `-O`, and `-Osize`) are faster than the SC versions at the 95% confidence level. For N = 2, SC with `-Osize` is faster than the GCD versions, while SC with `-O` is slower (see Figure 4.2a and Table 4.1). The difference between SC `-Onone` and the GCD versions is not statistically significant at the 95% confidence level.

For N = 10, all SC versions have lower average execution times than the GCD versions, although the difference between GCD `-Osize` and SC `-Onone` is not statistically significant (see Figure 4.2b and Table 4.2).

The memory consumption profile did not differ with compilation flag,




(a) Concentrated at data point with 2 tasks created.

(b) Concentrated at data point with 10 tasks created.

Figure 4.2: Execution time of the *SpawnManyWaiting* benchmark.

Table 4.1: Difference (GCD - SC) between the means of the execution time results in *SpawnManyWaiting* for N = 2.

| | **SC (-Onone)** | **SC (-O)** | **SC (-Osize)** |
|---|---|---|---|
| **GCD (-Onone)** | 1.180e-03 ± 1.368e-03 | -4.500e-03 ± 9.952e-04 | 2.286e-03 ± 1.343e-03 |
| **GCD (-O)** | 5.665e-04 ± 1.397e-03 | -5.114e-03 ± 1.035e-03 | 1.673e-03 ± 1.372e-03 |
| **GCD (-Osize)** | 6.176e-04 ± 1.410e-03 | -5.062e-03 ± 1.053e-03 | 1.724e-03 ± 1.386e-03 |

Table 4.2: Difference (GCD - SC) between the means of the execution time results in *SpawnManyWaiting* for N = 10.

| | **SC (-Onone)** | **SC (-O)** | **SC (-Osize)** |
|---|---|---|---|
| **GCD (-Onone)** | 5.669e-03 ± 5.380e-03 | 1.404e-02 ± 4.125e-03 | 1.274e-02 ± 4.413e-03 |
| **GCD (-O)** | 5.720e-03 ± 5.384e-03 | 1.409e-02 ± 4.130e-03 | 1.279e-02 ± 4.417e-03 |
| **GCD (-Osize)** | 5.272e-03 ± 5.404e-03 | 1.364e-02 ± 4.156e-03 | 1.235e-02 ± 4.442e-03 |

so only the `-Onone` versions are included. Figure 4.3 shows the memory consumption profile for the GCD version, and Figure 4.4 for the SC version. Overall, the memory consumption profile was stable at just under 14 MB of memory used.

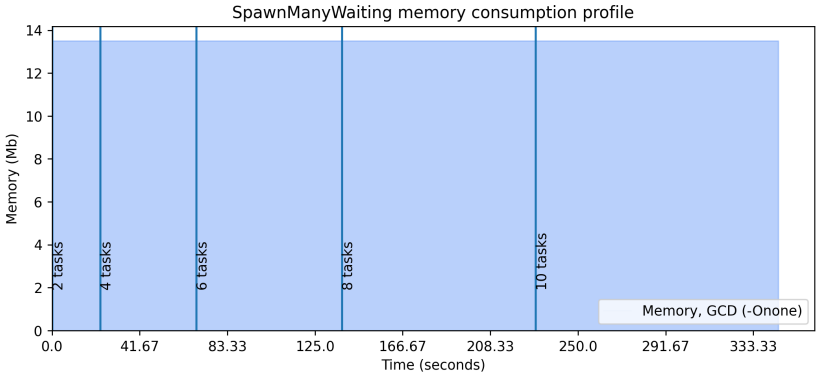



Figure 4.3: Memory consumption profile for the GCD version, `-Onone`, of the *SpawnManyWaiting* benchmark.

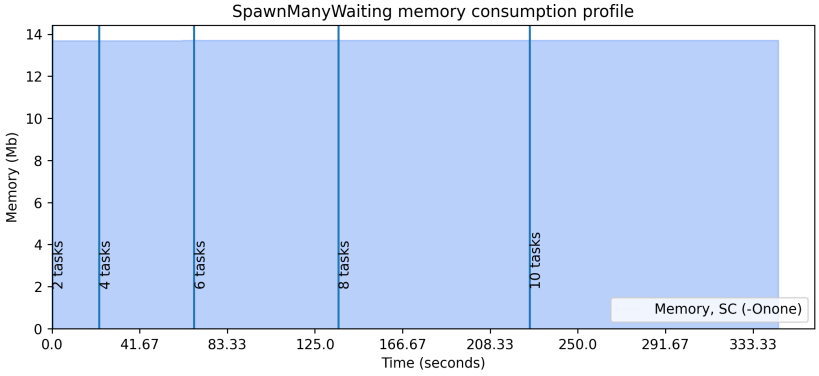



Figure 4.4: Memory consumption profile for the SC version, `-Onone`, of the *SpawnManyWaiting* benchmark.

## 4.2 The *SpawnManyWaitingGroup* benchmark

Figure 4.5 shows the execution time of the *SpawnManyWaitingGroup* benchmark as a function of number of tasks created, $N$. Table 4.3 reports approximate 95% confidence intervals for the differences between the means of the execution times for N = 50 (tables for all values of N are available in Appendix A as Tables A.6-A.11).

The difference between the two concurrency models starts out small, but increases monotonically as the benchmark parameter is increased. The execution times of the GCD versions increase steadily, while the execution times of the SC versions remain constant.

For N = 50, the difference between SC `-Osize` and the GCD versions

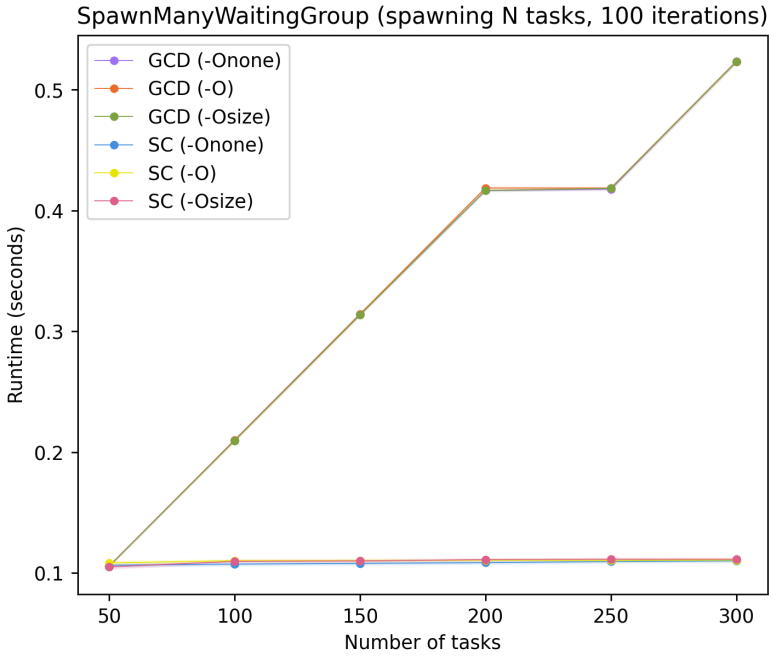



Figure 4.5: Overview of execution time for the *SpawnManyWaitingGroup* benchmark with all benchmark parameters.

is not statistically significant (see Table 4.3), but for all other differences across all other N values, the differences are statistically significant at the 95% confidence level.

Table 4.3: Difference (GCD - SC) between the means of the execution time results in *SpawnManyWaitingGroup* for N = 50.

| | **SC (-Onone)** | **SC (-O)** | **SC (-Osize)** |
|---|---|---|---|
| **GCD (-Onone)** | -7.928e-04 ± 6.780e-04 | -2.758e-03 ± 7.586e-04 | 6.182e-04 ± 8.638e-04 |
| **GCD (-O)** | -8.285e-04 ± 6.770e-04 | -2.794e-03 ± 7.577e-04 | 5.825e-04 ± 8.630e-04 |
| **GCD (-Osize)** | -8.263e-04 ± 6.773e-04 | -2.791e-03 ± 7.580e-04 | 5.847e-04 ± 8.632e-04 |

The memory consumption profile did not differ with compilation flag, so only the `-Onone` versions are included. Figure 4.6 shows the memory consumption profile for the GCD version, and Figure 4.7 for the SC version. Overall, the memory consumption profile was stable at around 15 MB of memory used. There is a minor difference between the GCD and SC version, with the former consuming slightly more memory overall than the latter (around 0.5 MB). Furthermore, the memory consumption increases by a small

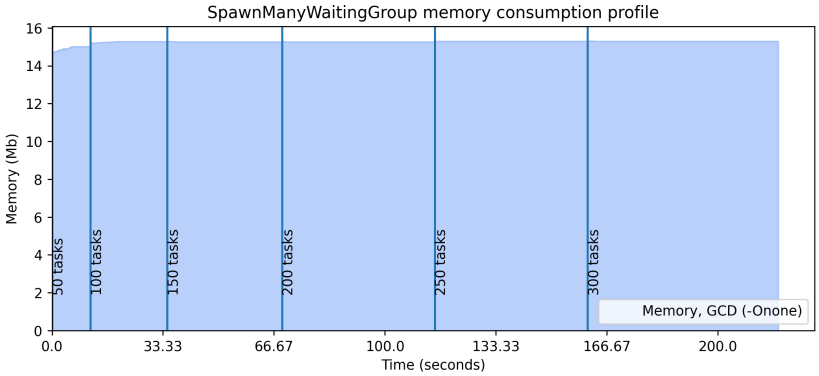



Figure 4.6: Memory consumption profile for the GCD version, `-Onone`, of the *SpawnManyWaitingGroup* benchmark.

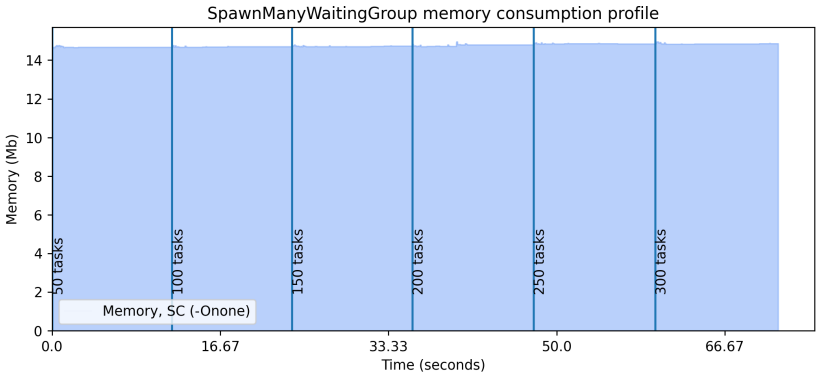



Figure 4.7: Memory consumption profile for the SC version, `-Onone`, of the *SpawnManyWaitingGroup* benchmark.

amount in the GCD version when N is increased from 50 to 100, an increase which is not seen in the SC version.

## 4.3 The *SpawnManyActors* benchmark

Figure 4.8 shows the execution time of the *SpawnManyActors* benchmark as a function of number of actor instances created, $N$. Tables A.12-A.21 in Appendix A report approximate 95% confidence intervals for the differences between the means of the execution times.

The difference between the two concurrency models is apparent for all values of N, with the SC versions consistently outperforming the GCD versions. All differences are statistically significant at the 95% confidence

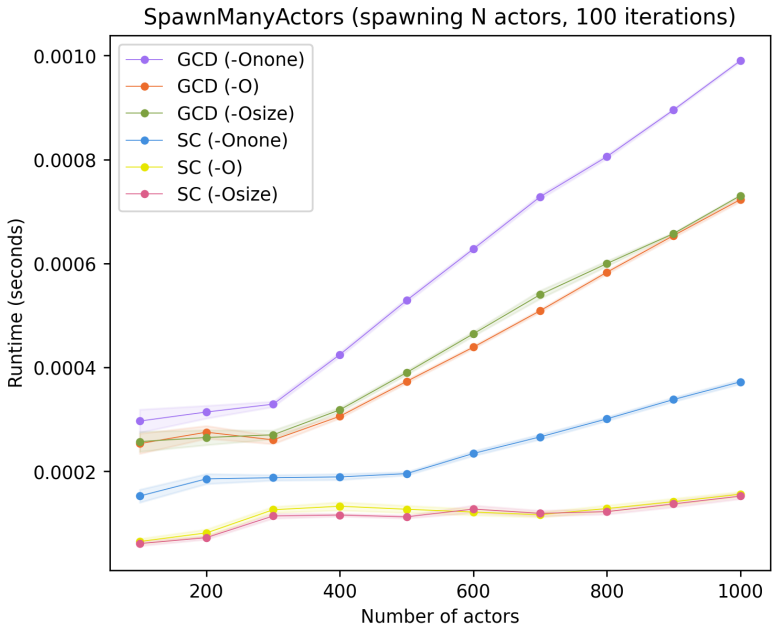



Figure 4.8: Overview of execution time for the *SpawnManyActors* benchmark with all benchmark parameters.

level, and keep increasing as N increases.

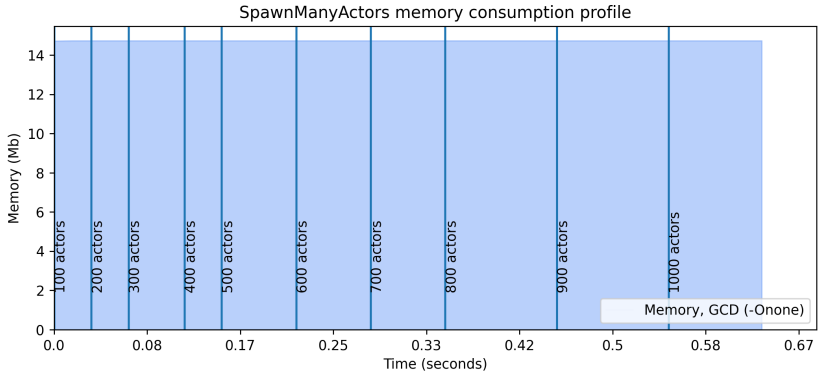



Figure 4.9: Memory consumption profile for the GCD version, `-Onone`, of the *SpawnManyActors* benchmark.

The memory consumption profile did not differ with compilation flag, so only the `-Onone` versions are included. Figure 4.9 shows the memory consumption profile for the GCD version, and Figure 4.10 for the SC version. Overall, the memory consumption profile was stable at around 15 MB of memory used for both versions.

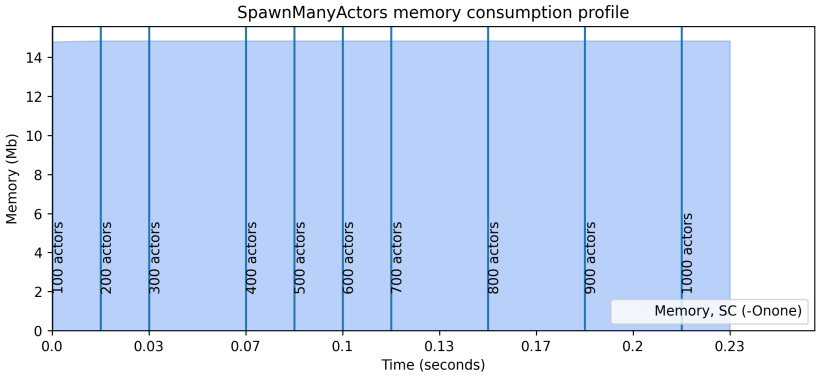



Figure 4.10: Memory consumption profile for the SC version, `-Onone`, of the *SpawnManyActors* benchmark.

## 4.4 The *Fibonacci* benchmark

Figure 4.11 shows the execution time of the *Fibonacci* benchmark as a function of the Fibonacci number calculated, $N$. Table 4.4 reports approximate 95% confidence intervals for the differences between the means of the execution times for N = 25 (tables for all values of N are available in Appendix A as Tables A.22-A.26).

In this benchmark, the difference between the two concurrency models is small, but mostly statistically significant at the 95% confidence level, for lower values of N. As N increases, so does the difference, and at N = 25 all three SC versions are outperformed by the GCD versions, and the differences are statistically significant.

Table 4.4: Difference (GCD - SC) between the means of the execution time results in *Fibonacci* for N = 25.

| | **SC (-Onone)** | **SC (-O)** | **SC (-Osize)** |
|---|---|---|---|
| **GCD (-Onone)** | -1.293e-01 ± 6.152e-04 | -4.470e-02 ± 5.859e-04 | -4.497e-02 ± 6.063e-04 |
| **GCD (-O)** | -1.375e-01 ± 6.176e-04 | -5.293e-02 ± 5.884e-04 | -5.319e-02 ± 6.087e-04 |
| **GCD (-Osize)** | -1.358e-01 ± 6.370e-04 | -5.122e-02 ± 6.088e-04 | -5.148e-02 ± 6.285e-04 |

The memory consumption profile in the SC versions did not differ much between the `-O` and `-Osize` flags, but it did differ for these two flags compared to the `-Onone` flag. Hence, the `-Onone` and `-O` versions are included, in Figure 4.12 and Figure 4.13, respectively. The memory consumption profile for these is characterized by large spikes in memory consumption, interleaved

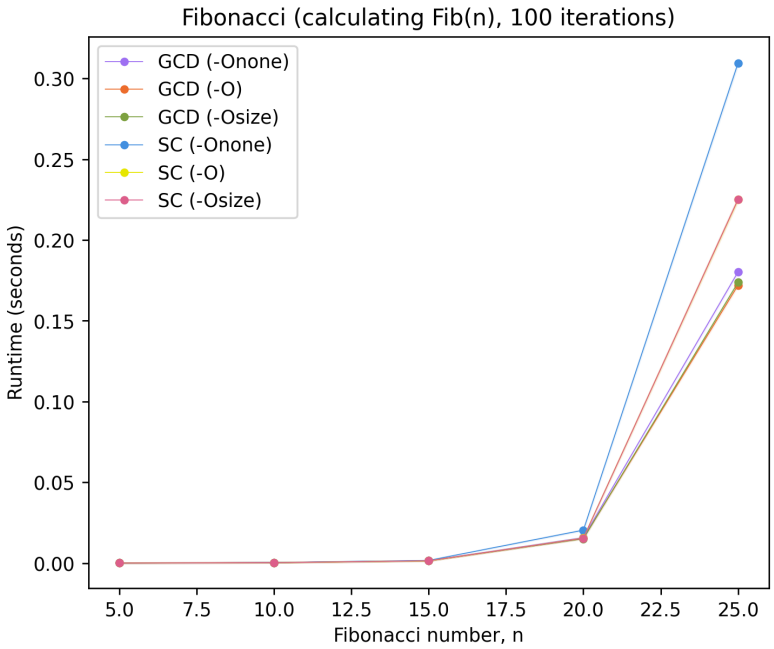



Figure 4.11: Overview of execution time for the *Fibonacci* benchmark with all benchmark parameters.

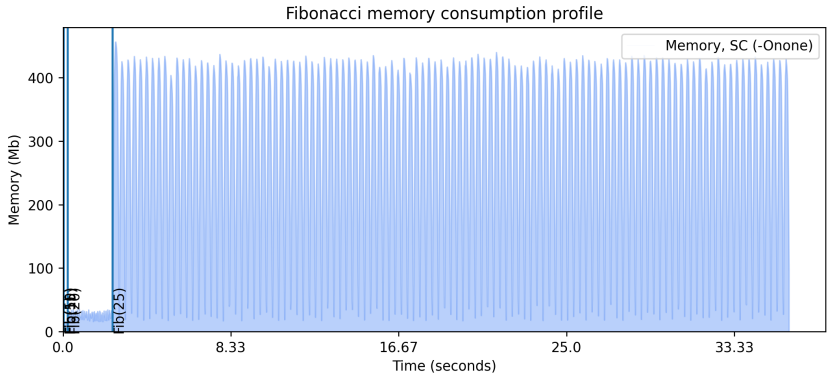



Figure 4.12: Memory consumption profile for the SC version, `-Onone`, of the *Fibonacci* benchmark.

by low consumption in between. Interestingly, the SC version with `-Onone` show memory peaks of more than 400 MB when calculating `Fib(25)`, while the SC version with `-O` show peaks of around 250 MB for the same parameter.

Even at lower Fibonacci numbers, the memory usage of SC `-Onone` is much higher than that of SC `-O`. For example, when calculating `Fib(20)`, SC `-Onone` peaks at almost 50 MB while SC `-O` peaks at around 25 MB.

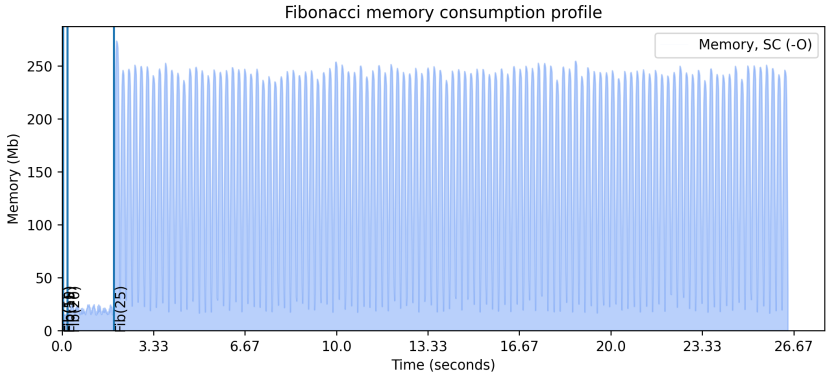



Figure 4.13: Memory consumption profile for the SC version, `-O`, of the *Fibonacci* benchmark.

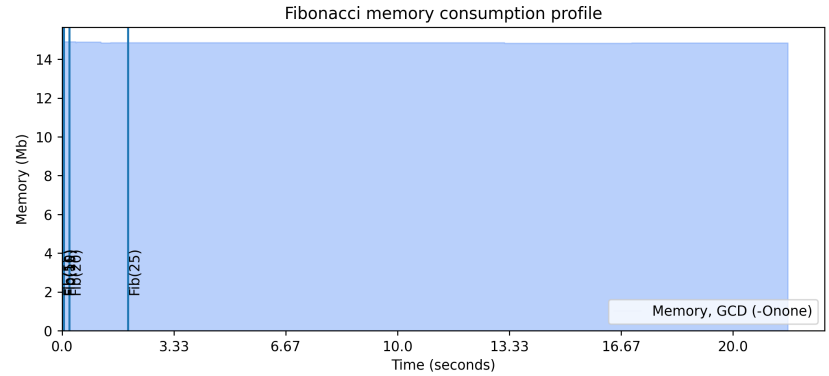



Figure 4.14: Memory consumption profile for the GCD version, `-Onone`, of the *Fibonacci* benchmark.

For the GCD versions, the memory consumption profile did not differ with compilation flag, so only the `-Onone` version is included. Figure 4.14 shows the memory consumption profile for the GCD version. In this version, the memory consumption was stable at around 15 MB of memory used.

## 4.5 The *NQueens* benchmark

Figure 4.15 shows the execution time of the *NQueens* benchmark as a function of the number of queens placed, $N$. Table 4.5 reports approximate 95% confidence intervals for the differences between the means of the execution times for N = 5 (tables for all values of N are available in Appendix A as Tables A.27-A.31).

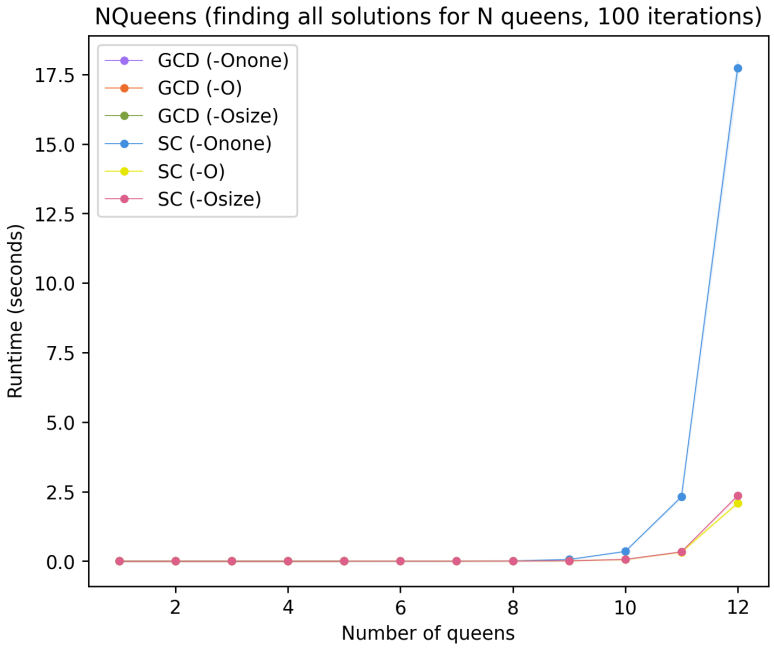



Figure 4.15: Overview of execution time for the *NQueens* benchmark with all benchmark parameters.

In this benchmark, the GCD versions stalled when calculating anything more than N = 5 (a possible reason for this will be discussed in the next chapter). The SC versions were able to run with a maximum of N = 12. After this point, the app ran out of memory and crashed.

The difference between the two concurrency models is small but statistically significant at the 95% confidence level for all values of N. The SC versions consistently outperform the GCD versions for the N values where both versions run (i.e. for N = 1 up to N = 5). This is not visible in Figure 4.15 because the differences at N = 5 and below were so small compared to the runtime of the higher N values where only the SC version ran. At N = 12 there is a large difference between SC `-Onone` and the other two SC versions.

Table 4.5: Difference (GCD - SC) between the means of the execution time results in *NQueens* for N = 5.

| | **SC (-Onone)** | **SC (-O)** | **SC (-Osize)** |
|---|---|---|---|
| **GCD (-Onone)** | 3.855e-03 ± 2.104e-03 | 3.981e-03 ± 2.104e-03 | 3.990e-03 ± 2.104e-03 |
| **GCD (-O)** | 1.905e-03 ± 1.452e-03 | 2.030e-03 ± 1.452e-03 | 2.039e-03 ± 1.452e-03 |
| **GCD (-Osize)** | 2.352e-03 ± 1.719e-03 | 2.478e-03 ± 1.719e-03 | 2.487e-03 ± 1.719e-03 |

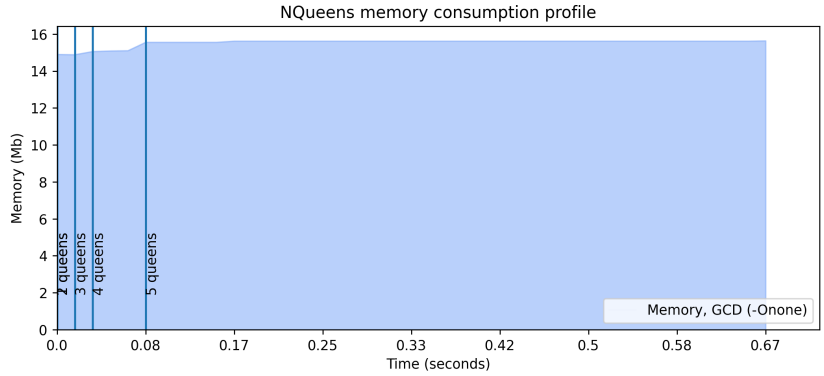



Figure 4.16: Memory consumption profile for the GCD version, `-Onone`, of the *NQueens* benchmark.

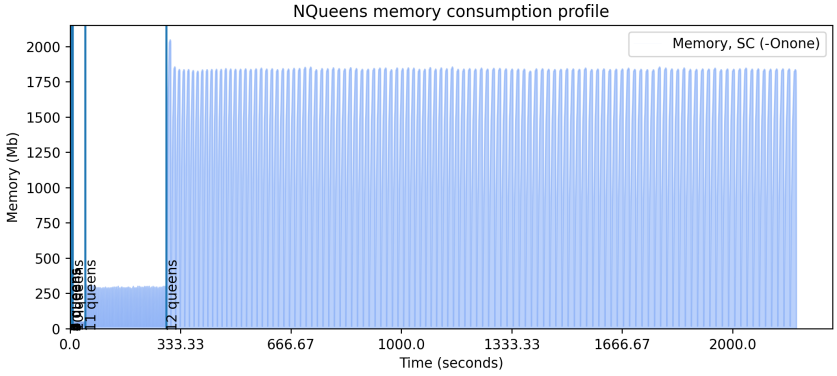



Figure 4.17: Memory consumption profile for the SC version, `-Onone`, of the *NQueens* benchmark.

The memory consumption profile did not differ with compilation flag, so only the `-Onone` versions are included. Figure 4.16 shows the memory consumption profile for the GCD version, and Figure 4.17 for the SC version. The memory consumption profile for the SC versions is similar to that of the *Fibonacci* benchmark, with large spikes in memory consumption, interleaved

by low consumption in between. For comparison with the GCD version, which only ran up to N = 5, Figure 4.18 shows the memory consumption profile at N = 5 of SC `-Onone`. In both the GCD and SC versions, the memory consumption is relatively stable at around 15 MB for N values of 5 and below.

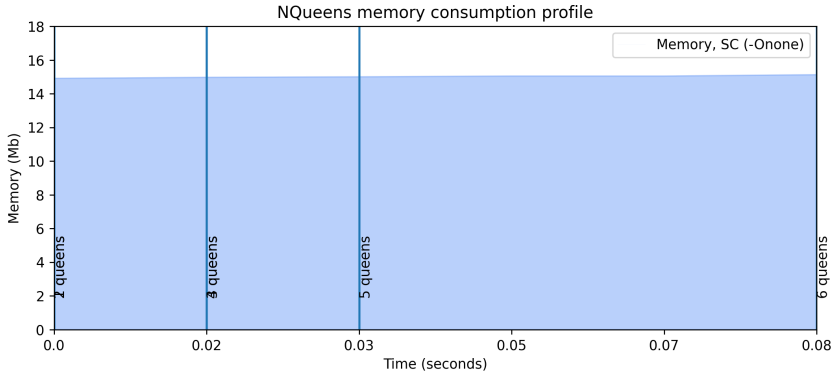



Figure 4.18: Memory consumption profile for the SC version, `-Onone`, of the *NQueens* benchmark, at N = 5.

## 4.6 The *MatrixMultiplication* benchmark

Figure 4.19 shows the execution time of the *MatrixMultiplication* benchmark as a function of the dimensions of the square matrices, $N \times N$. Table 4.6 reports approximate 95% confidence intervals for the differences between the means of the execution times for N = 25 (tables for all values of N are available in Appendix A as Tables A.32-A.35).

In this benchmark, all differences for all values of N are statistically significant. GCD with `-Onone` is the slowest version, for all values of N. The second slowest version is SC `-Onone`, which is outperformed by GCD `-O` and GCD `-Osize`. The fastest versions are SC `-Osize` and SC `-O`. The differences in execution time increase as the benchmark parameter is increased.

The memory consumption profile in the GCD versions did not differ much between the `-O` and `-Osize` flags, but it did differ for these two flags compared to the `-Onone` flag. Hence, the `-Onone` and `-O` versions are included, in Figure 4.20 and Figure 4.21, respectively. The memory consumption profile for these is relatively stable within the different benchmark parameter regions. The consumption is increased by slightly less than 1 MB when the benchmark parameter is increased. The memory consumption of GCD `-Onone` is

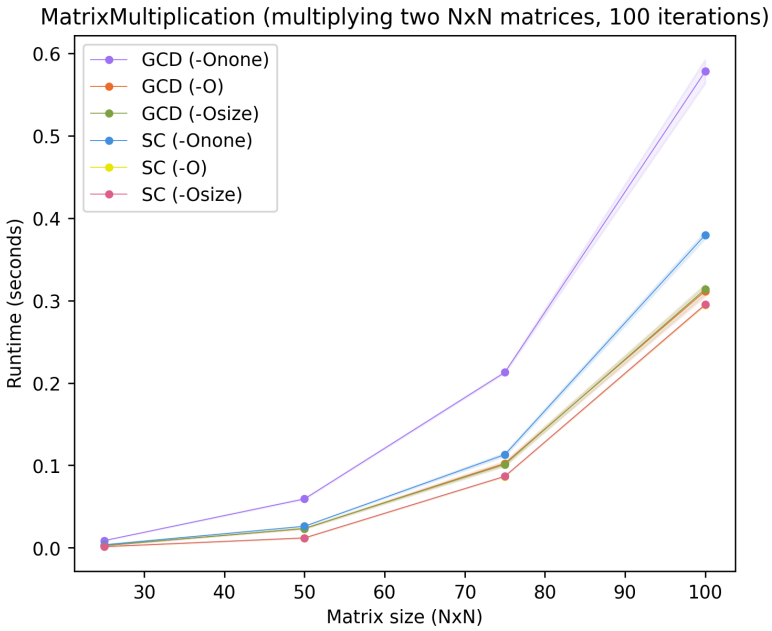



Figure 4.19: Overview of execution time for the *MatrixMultiplication* benchmark with all benchmark parameters.

Table 4.6: Difference (GCD - SC) between the means of the execution time results in *MatrixMultiplication* for N = 25.

| | **SC (-Onone)** | **SC (-O)** | **SC (-Osize)** |
|---|---|---|---|
| **GCD (-Onone)** | 5.044e-03 ± 8.831e-05 | 7.276e-03 ± 1.187e-04 | 7.297e-03 ± 1.101e-04 |
| **GCD (-O)** | -9.945e-04 ± 5.282e-05 | 1.237e-03 ± 9.533e-05 | 1.258e-03 ± 8.438e-05 |
| **GCD (-Osize)** | -9.464e-04 ± 5.244e-05 | 1.285e-03 ± 9.512e-05 | 1.306e-03 ± 8.414e-05 |

just over 1 MB lower for each respective benchmark parameter than the corresponding consumption in GCD `-O` (and GCD `-Osize`, not pictured).

The memory consumption profile in the SC versions did differ for all three flags, and thus all three versions are included, in Figures 4.22-4.24. The profile in SC `-Onone` has a large spike at N = 75, of roughly 20 MB. In general, the memory consumption profiles of all three SC versions are characterized by spikes in memory consumption, although these are slightly higher for the `-Onone` and `-Osize` versions.

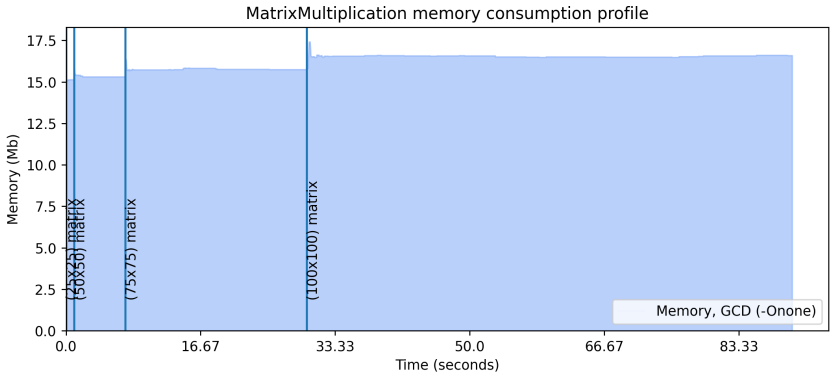



Figure 4.20: Memory consumption profile for the GCD version, `-Onone`, of the *MatrixMultiplication* benchmark.

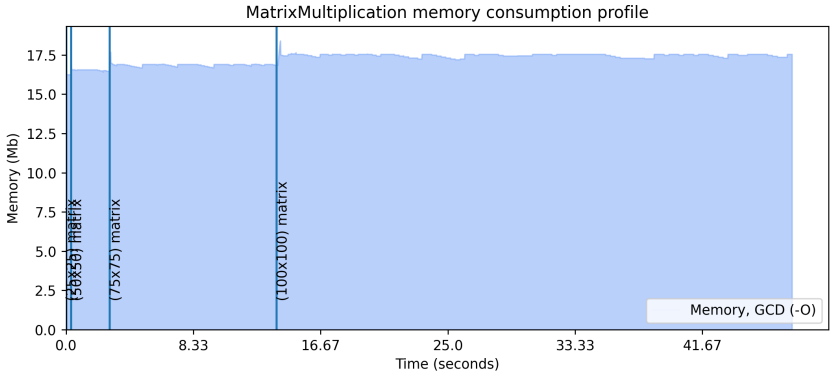



Figure 4.21: Memory consumption profile for the GCD version, `-O`, of the *MatrixMultiplication* benchmark.

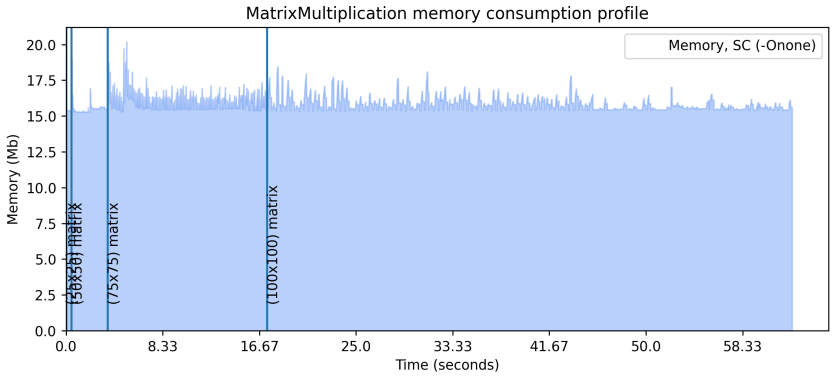



Figure 4.22: Memory consumption profile for the SC version, `-Onone`, of the *MatrixMultiplication* benchmark.

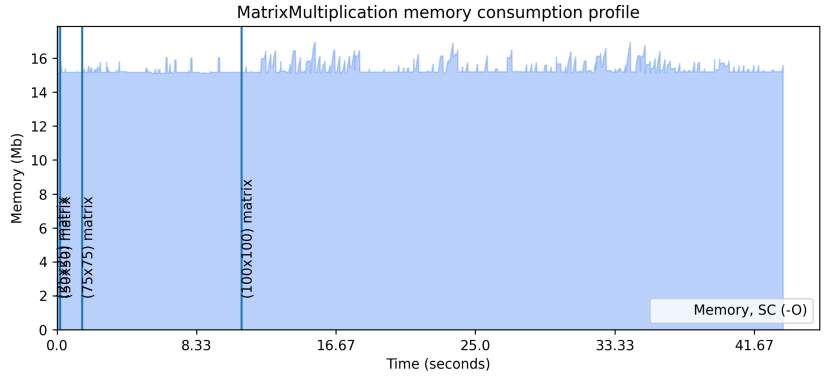



Figure 4.23: Memory consumption profile for the SC version, `-O`, of the *MatrixMultiplication* benchmark.

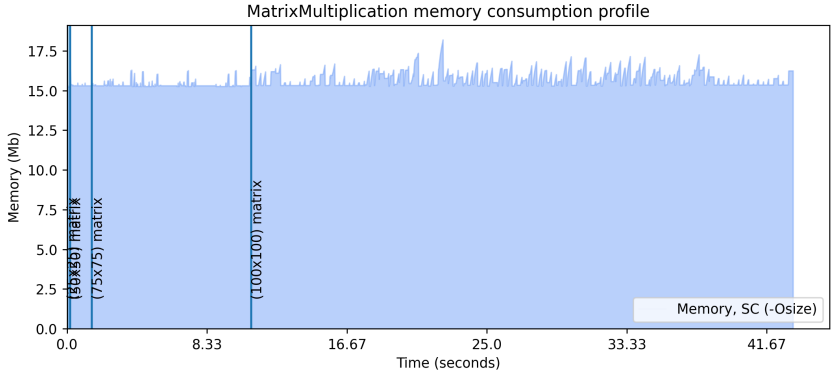



Figure 4.24: Memory consumption profile for the SC version, `-Osize`, of the *MatrixMultiplication* benchmark.

# Chapter 5

# Discussion

This chapter contains a discussion of the results presented in the previous chapter. The results of each benchmark are analysed, followed by a discussion of various aspects of the thesis such as the data analysis methodology and choice and implementation of benchmarks.

## 5.1 Base benchmarks

The results of the *SpawnManyWaiting* benchmark indicate that there is no major overhead for creating and running Swift concurrency `Tasks` compared to dispatching tasks to queues in GCD. The GCD versions did outperform the SC versions for some benchmark parameters, but the differences were generally small, and both models appear to scale similarly as the number of tasks increase. Because the benchmark creates tasks that wait, the amount of tasks created was kept small, to keep the total runtime of the benchmark within reasonable limits. That said, creating more tasks would likely have magnified the (possible) effects of task creation overhead, and could have been done if there was more time.

When running the *SpawnManyWaitingGroup* benchmark, more tasks could be created, since these ran concurrently. The results of that benchmark indicate that Swift concurrency's `TaskGroup` has better performance than GCD's `DispatchGroup`. The steep increase in runtime for the GCD version could potentially be explained by the problem of "thread explosion", as discussed in Section 1.2.1. The (GCD) tasks enqueued on the dispatch queue do block when they invoke `Thread.sleep()`, which is exactly the scenario where thread explosion might occur. It is therefore interesting that there is no notable difference in memory consumption profile between the two

concurrency models, because a thread explosion is supposed to impact not only execution time but also memory usage. It is unclear why this effect does not show in the memory usage. One explanation might be that the runtime can optimize the code of the *SpawnManyWaitingGroup* benchmark, since the tasks do not perform any real work. As mentioned in Section 1.5, it was too time-consuming to automatically measure and report thread creation, but this is an example where it would have been useful to measure.

In *SpawnManyActors*, there is no discernible difference in memory consumption, but there is a difference in execution time, with Swift concurrency consistently outperforming GCD. This indicates that Swift concurrency actors are more than just an "alternative way" of creating a class with a private serial dispatch queue for synchronization. The compiler and/or runtime can seemingly make some optimizations for actors in Swift concurrency that it cannot make in the GCD version. One explanation for this could be that in the Swift concurrency model, the actors and their semantics are part of the language, and can therefore be reasoned about by the compiler in more detail than is possible in the GCD model. To investigate this further, one might use compiler flags to output the different phases of the intermediate representation (IR) of the Swift code as it is compiled [77], to examine any possible differences in the generated low-level code.

## 5.2 Applied benchmarks

The *Fibonacci* benchmark shows Swift concurrency outperformed by GCD to the point where Swift concurrency running with optimizations is still slower than GCD running without. Most of the difference in execution time can probably be attributed to the large difference in memory consumption. The spiky memory behaviour of the Swift concurrency version indicates that a lot of memory allocations and deallocations take place within each iteration, which has a performance overhead. Why the usage peaks at more than 200 MB more for SC than GCD is not obvious. It could simply be that the implementation of `async let` tasks still needs more tuning in the language to reach adequate performance. Alternatively, it might be the recursion that affects the execution time and memory consumption negatively. The choice of including recursive benchmarks was based on the review of related work, but highly recursive fine-grained concurrent implementations might not have been the main intended use case for the new model.

In the execution times for the *NQueens* benchmark, all SC versions outperformed all GCD versions for the benchmark parameters where both

models ran ($N \leq 5$). The memory consumption profile for $N \leq 5$ was highly similar for both models. The fact that the memory consumption was similar, and low, could indicate that recursion was not problematic in this case. However, the memory consumption did increase greatly for larger values of N, especially N = 11 and N = 12. In any case, since the GCD version stalled at N = 5, it is unclear what the memory consumption would have looked like for GCD at larger values of N, so it is difficult to draw any conclusions from this aspect of the results.

The reason for why the GCD version stalled at N > 5 is suspected to be the alleged limit of 64 threads created per concurrent dispatch queue in the GCD model, as mentioned in Section 2.3. Because each recursive call to `solutionsGCD` creates a new `DispatchGroup` and blocks the thread until all child tasks in the group have finished, it is hypothesized that the program is stalled with 64 blocked threads (created to service the global concurrent dispatch queue of default quality-of-service). All 64 threads would be waiting for some child tasks to complete, which will not happen because these child tasks are stuck on the queue and cannot be executed because no new threads can be created to service them. While this is not strictly related to the performance of the underlying concurrency model, it highlights an interesting consequence of how the GCD model works. In the SC model, there is no limit to the amount of `Tasks` that can be created and executed (because there is no risk for thread explosion), and so the problem does not appear.

Finally, in the *MatrixMultiplication* benchmark, the SC model outperformed the GCD model in terms of execution time for corresponding optimization flags. Both models scale similarly, but the difference in runtime does seem to increase as the size of the matrices increase. Part of the difference in execution time can likely be attributed to the use of a private serial dispatch queue in the GCD version, which is replaced by an actor type in the SC version. In that regard, the results of this benchmark are congruent with those seen for the *SpawnManyActors* benchmark.

The memory consumption profiles for this benchmark show some differences between the two models. The memory consumption profile in the GCD versions is relatively stable unlike the profile of the SC versions, which feature spikes in the memory consumption. These spikes are also of varying height, indicating that the memory consumption of each individual iteration fluctuates over time. In that sense, the spikes differ from those observed in the *Fibonacci* and *NQueens* benchmarks, which were relatively equal in height. This difference might arise from the loop-like structure of the benchmark. Because all child tasks are created at once, the scheduling of them can vary

to a greater degree than is possible for the recursive applied benchmarks. In those benchmarks, only parts of the total amount of work items can be started immediately; the others are started only once some other work items have finished.

## 5.3 General remarks

One major drawback with the chosen methodology of evaluating the performance by means of artificial benchmarks is that they will not necessarily be able to accurately model what the performance of "real" applications could look like. It would be interesting to develop more realistic, full-scale applications based on the two models and compare them. The difficulty with such an approach would instead be to ensure that the two models are afforded "equal opportunities to shine". This trade-off needed to be considered already for the benchmarks of this thesis, especially the applied ones. The designs of the two concurrency models differ, and there is a trade-off between comparing pair-wise implementations that are strictly semantically equivalent versus comparing more idiomatic implementations that achieve the same end result, but with slightly different semantics. In this trade-off, the base benchmarks tend towards the former while the applied benchmarks tend towards the latter.

Furthermore, the differences in the benchmark results for the two models may not only come from how the models themselves are implemented, but might also arise due to other factors. For example, it might be because the implementations did not represent the intended usage of the primitives, i.e. be explained by lack of experience of the implementer. It could also be because the implementations did not represent the intended use cases of the primitives. The Swift programming language is, as mentioned in Chapter 2, mainly used to program applications for iOS. It could be argued that calculating large Fibonacci numbers, for instance, does not represent a typical scenario for an iOS developer. This is another case for why implementing a larger, "more typical" application might have been suitable.

To allow for even more analysis, it could also have been beneficial to not only include *wall clock time* but also *CPU time* in the execution time measurements. Since it is not clear whether there would be a difference in how other system effects affect a running Swift program depending on the concurrency model used, separating these measurements would make it easier to see whether the same results can be seen for both measurement methods or if they are different. A drawback with this could be that both values would have to be measured simultaneously (to allow for comparison), which likely

might affect their results since the measurement of one could affect the results of the other, and vice versa.

When it comes to the analysis of the data, the sample size of 100 iterations was assumed to be large enough to ensure that the mean of the samples would be approximately normally distributed, as per the Central Limit Theorem presented in Section 2.5.2. However, Chen et al [78] argue that it is unclear how many samples are needed to ensure that the CLT can be used, and conduct an experiment where they find that several hundred samples are needed to get an approximately normal distribution. Both Chen et al [78] and Hoefler and Belli [79] therefore suggest performing normality tests on the sample data to test for normality. This is also something that could have been done to increase the validity of the results.

# Chapter 6

# Conclusions

This thesis aimed to investigate the differences in execution time and memory consumption between two common concurrency models on the iOS platform: *Grand Central Dispatch* (GCD) and *Swift concurrency*. The differences were evaluated by measuring execution time and memory usage over six benchmarks; three microbenchmarks (the *base benchmarks*) that exercised the concurrency primitives, and three application benchmarks (the *applied benchmarks*) inspired by previous work.

In general, differences in execution time between the two models were statistically significant, but no model consistently outperformed the other in execution time nor memory consumption across all benchmarks. The Swift concurrency versions outperformed the GCD versions in execution time for two of the three base benchmarks. For the third one, *SpawnManyWaiting*, there were some small, but statistically significant differences in both directions (i.e. GCD was sometimes better than Swift concurrency, and vice versa), and the scaling was identical. The applied benchmarks showed some cases where the Swift concurrency model was outperformed by the GCD concurrency model. This was especially the case in the *Fibonacci* benchmark, but also for some optimization flags of the *MatrixMultiplication* benchmark. In the *NQueens* benchmark, however, Swift concurrency outperformed GCD, and was also able to run with a higher benchmark parameter before stalling. In this regard, the results of this work indicate that neither model is necessarily "better" than the other from a performance standpoint. This implies that decisions of which model to adopt can be based on other aspects such as ease-of-use of the models, business requirements, or other considerations.

The performance evaluation could have been made more comprehensive by including more performance metrics such as thread creation and perhaps

looking in more detail at memory allocations; this could constitute an interesting avenue for future work. A general insight when performing the evaluation was that performance measurements on the highly sandboxed iOS platform is not straightforward. The *Instruments* tool, part of Apple's integrated development environment *Xcode*, is usually recommended as a means to get detailed information on various metrics. However, this information was complicated to export and manipulate outside of the *Instruments* tool, and the widespread usage of the tool made it difficult to find alternative ways to measure performance.

Overall, the objectives outlined in Section 1.3.1 are deemed as fulfilled. The obtained measurement results showed differences in execution time and memory consumption between the two concurrency models, and we consider the research question as answered.

## 6.1 Future work

Although some suggestions for future work have been hinted at throughout the thesis, especially in Chapters 1 and 5, this section goes into more detail and proposes some ways to build upon the work of this thesis.

Firstly, it would be valuable to conduct experiments with more benchmarks of different categories and extents (i.e. both small-scale and large-scale), than those suggested in this work. As mentioned previously, it would also be of interest to construct more realistic mobile applications and benchmark them in their entirety as opposed to looking at small, artificial benchmarks as was done for this study.

It could also be interesting to see how the performance differs when varying some of the parameters that were kept constant in the experiments of this thesis. Varying the device on which the experiments run could be one possibility; it is likely that the hardware configuration would influence results to at least some extent. Running benchmarks with other applications open in the background could be another way to increase realism in the experiments, since this is probably a common scenario when apps run in the real world.

Slightly orthogonal to the work of this thesis, but yet interesting and worthwhile to investigate, would be methods with which to conduct performance evaluations on the iOS platform, to ensure reliability and reproducibility. As mentioned in the previous section, this is already made complicated by the sandboxed nature of the iOS operating system, but may be complicated further still by other factors not considered in this thesis. For example, Mytkowicz et al. found that seemingly innocuous changes to environment variables

and program linking order can affect performance greatly and introduce measurement bias in performance evaluations of computer systems [80]. Unfortunately, research on performance analysis in smartphone environments is scarce. By developing a rigorous methodology for performance evaluations on smart devices, the cohesiveness of future work could be improved upon, and the ad hoc nature of adopted methodologies could be reduced.

Lastly, this thesis focused on empirical analysis of the two models by running experiments to compare them. It could also be interesting to carry out an analysis from a more analytic perspective, e.g. by examining the low-level generated code to see how it differs between the two models and drawing conclusions from that.

## 6.2 Reflections on economic, social, and environmental sustainability aspects of the work

Smartphones are increasingly commonplace and each year new models are released by major manufacturers, such as Apple and Samsung. If the software on these devices is optimized for performance and efficiency, it could hopefully lead to a decreased need to buy new models as soon as they come out, and also a decreased need to buy the best and shiniest model. Among other things, this would be more inclusive towards those who may not afford to spend the relatively large amount of money that purchasing a smartphone entails. It would also have a direct effect on environmental sustainability, since the manufacturing of the devices requires materials whose extraction pose a negative impact on the environment. This can be related to the Sustainable Development Goals (SDGs) 9 and 12 [81, 82].

Regarding ethical aspects, the work compares alternatives to construct software with the intention to provide useful information that can aid decision making. There could be drawbacks depending on what the information is used for, but this is hard to predict. In general, studying programming language implementation should probably be considered as neither ethical nor unethical; it is impossible to be held accountable for what ends the language itself is used for, whether they be good or bad.

The work and its results are relevant to the host company, Bontouch, whose principal business consists in creating mobile applications (such as for example the *Swish* and *PostNord* applications). By identifying differences between the two concurrency models, and especially advantages and drawbacks in each

model, better decisions can be made to ensure that applications can be highly performant while still utilizing the hardware as efficiently as possible. As mentioned in Chapter 1, it is also of interest to the academical community to gain insight into how differences in programming language design and implementation affect the final performance of programs constructed by that language.

# References

[1] H. Sutter, "The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software," Dec. 2004. [Online]. Available: http://www.gotw.ca/publications/concurrency-ddj.htm

[2] P. Thoman, P. Gschwandtner, and T. Fahringer, "On the Quality of Implementation of the C++11 Thread Support Library," in *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, Mar. 2015. doi: 10.1109/PDP.2015.33 pp. 94–98, iSSN: 2377-5750. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/7092705

[3] J. McCall and D. Gregor, "SE-0296 - Async/await." [Online]. Available: https://github.com/apple/swift-evolution/blob/main/proposals/0296-async-await.md

[4] "Whitepaper: The Benefits of Multiple CPU Cores in Mobile Devices," 2010. [Online]. Available: https://www.nvidia.com/content/PDF/tegra_white_papers/Benefits-of-Multi-core-CPUs-in-Mobile-Devices_Ver1.2.pdf

[5] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces, Chapter 26*, 1st ed. Arpaci-Dusseau Books, August 2018. [Online]. Available: https://pages.cs.wisc.edu/~remzi/OSTEP/threads-intro.pdf

[6] "Java API Reference – Class Thread." [Online]. Available: https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Thread.html

[7] Apple, "Thread | Apple Developer Documentation." [Online]. Available: https://developer.apple.com/documentation/foundation/thread

[8] "Concurrency Programming Guide," Dec. 2012. [Online]. Available: https://developer.apple.com/library/archive/documentation/

General/Conceptual/ConcurrencyProgrammingGuide/Introduction/Introduction.html#//apple_ref/doc/uid/TP40008091-CH1-SW1

[9] "`DispatchQueue` | Apple Developer Documentation." [Online]. Available: https://developer.apple.com/documentation/dispatch/dispatchqueue

[10] Apple, "Swift concurrency: Behind the scenes - WWDC21 - Videos." [Online]. Available: https://developer.apple.com/videos/play/wwdc2021/10254

[11] Swift Core Team, "Swift Concurrency Roadmap." [Online]. Available: https://forums.swift.org/t/swift-concurrency-roadmap/41611

[12] J. McCall, J. Groff, D. Gregor, and K. Malawski, "SE-0304 - Structured concurrency." [Online]. Available: https://github.com/apple/swift-evolution/blob/main/proposals/0304-structured-concurrency.md

[13] J. McCall, D. Gregor, K. Malawski, and C. Lattner, "SE-0306 - Actors." [Online]. Available: https://github.com/apple/swift-evolution/blob/main/proposals/0306-actors.md

[14] Apple, "Introducing Swift Distributed Actors." [Online]. Available: https://www.swift.org/blog/distributed-actors/

[15] Apple, "Introducing Swift Async Algorithms." [Online]. Available: https://www.swift.org/blog/swift-async-algorithms/

[16] "`concurrentPerform(iterations:execute:)` | Apple Developer." [Online]. Available: https://developer.apple.com/documentation/dispatch/dispatchqueue/2016088-concurrentperform

[17] "`detached(priority:operation:)` | Apple Developer." [Online]. Available: https://developer.apple.com/documentation/swift/task/3856788-detached

[18] "`Task` | Apple Developer." [Online]. Available: https://developer.apple.com/documentation/swift/task

[19] Apple, "Swift Has Reached 1.0 - Swift Blog." [Online]. Available: https://developer.apple.com/swift/blog/?id=14

[20] J. Kastrenakes, “Apple’s new programming language Swift is now open source,” Dec. 2015. [Online]. Available: https://www.theverge.com/2015/12/3/9842854/apple-swift-open-source-released

[21] J. Timmer, “A fast look at Swift, Apple’s new programming language,” Jun. 2014. [Online]. Available: https://arstechnica.com/gadgets/2014/06/a-fast-look-at-swift-apples-new-programming-language/

[22] Apple, “About Swift.” [Online]. Available: https://www.swift.org/about/

[23] “Swift & Objective-C 2019 - The state of Developer Ecosystem in 2019 Infographic.” [Online]. Available: https://www.jetbrains.com/lp/devecosystem-2019/swift-objc/

[24] Apple, “Swift - Apple Developer.” [Online]. Available: https://developer.apple.com/swift/

[25] Apple, “Automatic Reference Counting.” [Online]. Available: https://docs.swift.org/swift-book/LanguageGuide/AutomaticReferenceCounting.html

[26] Apple, “Swift Compiler.” [Online]. Available: https://www.swift.org/swift-compiler/

[27] “Swift Programming Language,” Feb. 2022, original-date: 2015-10-23T21:15:07Z. [Online]. Available: https://github.com/apple/swift/blob/496da9dda967e76664498c33fe6b66cc89a195eb/docs/SIL.rst#sil-in-the-swift-compiler

[28] Apple, “Closures.” [Online]. Available: https://docs.swift.org/swift-book/LanguageGuide/Closures.html

[29] Apple, “Swift 5.5 Released!” [Online]. Available: https://www.swift.org/blog/swift-5-5-released/

[30] C. Lattner, “Swift Concurrency Manifesto.” [Online]. Available: https://gist.github.com/lattner/31ed37682ef1576b16bca1432ea9f782

[31] “About Threaded Programming,” Jul. 2014. [Online]. Available: https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/Multithreading/AboutThreads/AboutThreads.html#//apple_ref/doc/uid/10000057i-CH6-SW2

[32] "Dispatch | Apple Developer Documentation." [Online]. Available: https://developer.apple.com/documentation/DISPATCH

[33] "Dispatch Queues – Concurrency Programming Guide," Dec. 2012. [Online]. Available: https://developer.apple.com/library/archive/documentation/General/Conceptual/ConcurrencyProgrammingGuide/OperationQueues/OperationQueues.html#//apple_ref/doc/uid/TP40008091-CH102-SW1

[34] "ios - Number of threads created by GCD? - Stack Overflow." [Online]. Available: https://stackoverflow.com/questions/7213845/number-of-threads-created-by-gcd

[35] "DispatchQueue 64 item limit? | Apple Developer Forums." [Online]. Available: https://developer.apple.com/forums/thread/129617

[36] "Grand Central Dispatch." [Online]. Available: https://web.archive.org/web/20090920043909/http://images.apple.com/macosx/technology/docs/GrandCentral_TB_brief_20090903.pdf

[37] A. W. Appel, *Compiling with Continuations*. Cambridge University Press, 1991. [Online]. Available: https://doi.org/10.1017/CBO9780511609619

[38] J. C. Reynolds, "The discoveries of continuations," *LISP and Symbolic Computation*, vol. 6, no. 3-4, pp. 233–247, Nov. 1993. doi: 10.1007/BF01019459. [Online]. Available: http://link.springer.com/10.1007/BF01019459

[39] K. Asai and O. Kiselyov, "Introduction to Programming with Shift and Reset," in *ACM SIGPLAN Continuation Workshop 2011*, 2011. [Online]. Available: http://pllab.is.ocha.ac.jp/~asai/cw2011tutorial/main-e.pdf

[40] A. L. D. Moura and R. Ierusalimschy, "Revisiting coroutines," *ACM Transactions on Programming Languages and Systems*, vol. 31, no. 2, pp. 6:1–6:31, Feb. 2009. doi: 10.1145/1462166.1462167. [Online]. Available: https://doi.org/10.1145/1462166.1462167

[41] C. D. Marlin, *Coroutines: A Programming Methodology, a Language Design and an Implementation – Chapter 1. Introduction*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1980, pp. 1–8. ISBN 978-3-540-38378-9. [Online]. Available: https://doi.org/10.1007/3-540-10256-6_1

[42] "Python Wiki - Generators." [Online]. Available: https://wiki.python.org/moin/Generators

[43] "PEP 255 – Simple Generators." [Online]. Available: https://www.python.org/dev/peps/pep-0255/

[44] LLVM, "2021 LLVM Dev Mtg "Asynchronous Functions in Swift"," Dec. 2021. [Online]. Available: https://www.youtube.com/watch?v=H_K-us4-K7s

[45] E. W. Dijkstra, "Letters to the editor: go to statement considered harmful," *Communications of the ACM*, vol. 11, no. 3, pp. 147–148, 1968. [Online]. Available: https://dl.acm.org/doi/pdf/10.1145/362929.362947

[46] T. Ito and M. Matsui, "A Parallel Lisp Language PaiLisp and its Kernel Specification," in *US/Japan Workshop on Parallel Lisp*. Springer, 1989, pp. 58–100. [Online]. Available: https://link.springer.com/chapter/10.1007/BFb0024150

[47] M. Sústrik, "Structured Concurrency," Feb. 2016. [Online]. Available: https://250bpm.com/blog:71/

[48] M. Sústrik, "libdill: Structured Concurrency for C," Mar. 2022. [Online]. Available: https://github.com/sustrik/libdill

[49] N. J. Smith, "Structured Concurrency," 4 2018. [Online]. Available: https://vorpus.org/blog/notes-on-structured-concurrency-or-go-statement-considered-harmful/

[50] "Coroutines basics." [Online]. Available: https://kotlinlang.org/docs/coroutines-basics.html

[51] Apple, "Explore structured concurrency in Swift - WWDC21 - Videos." [Online]. Available: https://developer.apple.com/videos/play/wwdc2021/10134/

[52] J. McCall, J. Groff, D. Gregor, and K. Malawski, "SE-0317 - `async let` bindings." [Online]. Available: https://github.com/apple/swift-evolution/blob/main/proposals/0317-async-let.md

[53] G. A. Agha, "Actors: A model of concurrent computation in distributed systems." Massachusetts Inst of Tech Cambridge Artificial Intelligence Lab, Tech. Rep., 1985.

[54] J. Armstrong, “Erlang,” *Communications of the ACM*, vol. 53, no. 9, pp. 68–75, 2010. [Online]. Available: https://dl.acm.org/doi/fullHtml/10.1145/1810891.1810910

[55] D. J. Lilja, *Measuring computer performance: a practitioner’s guide*. Cambridge university press, 2005.

[56] H. Hayslett and P. Murphy, “Chapter VIII - MORE TESTS OF HYPOTHESES,” in *Statistics*, H. Hayslett and P. Murphy, Eds. Butterworth-Heinemann, 1981, pp. 113–130. ISBN 978-0-7506-0481-9. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780750604819500127

[57] H. Hayslett and P. Murphy, “Chapter X - CONFIDENCE LIMITS,” in *Statistics*, H. Hayslett and P. Murphy, Eds. Butterworth-Heinemann, 1981, pp. 150–167. ISBN 978-0-7506-0481-9. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780750604819500140

[58] K. Chauhan, S. Kumar, D. Sethia, and M. N. Alam, “Performance Analysis of Kotlin Coroutines on Android in a Model-View-Intent Architecture pattern,” in *2021 2nd International Conference for Emerging Technology (INCET)*, May 2021. doi: 10.1109/INCET51464.2021.9456197 pp. 1–6. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9456197

[59] S. M. Imam and V. Sarkar, “Integrating task parallelism with actors,” *ACM SIGPLAN Notices*, vol. 47, no. 10, pp. 753–772, 2012. doi: 10.1145/2398857.2384671. [Online]. Available: https://doi.org/10.1145/2398857.2384671

[60] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, “X10: an object-oriented approach to non-uniform cluster computing,” in *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, ser. OOPSLA ’05. New York, NY, USA: Association for Computing Machinery, 2005. doi: 10.1145/1094811.1094852. ISBN 978-1-59593-031-6 pp. 519–538. [Online]. Available: https://doi.org/10.1145/1094811.1094852

[61] Apple, “About Swift.” [Online]. Available: https://docs.swift.org/swift-book/

[62] Apple, “Meet async/await in Swift - WWDC21 - Videos.” [Online]. Available: https://developer.apple.com/videos/play/wwdc2021/10132/

[63] Apple, “Protect mutable state with Swift actors - WWDC21 - Videos.” [Online]. Available: https://developer.apple.com/videos/play/wwdc2021/10133

[64] “Recommended way to measure time in Swift? - Using Swift - Swift Forums.” [Online]. Available: https://forums.swift.org/t/recommended-way-to-measure-time-in-swift/33326

[65] “`mach_absolute_time` | Apple Developer Documentation.” [Online]. Available: https://developer.apple.com/documentation/kernel/1462446-mach_absolute_time

[66] “Inside M1 Macs: Time and logs – The Eclectic Light Company.” [Online]. Available: https://eclecticlight.co/2020/11/27/inside-m1-macs-time-and-logs/

[67] “Task Information - The GNU Mach Reference Manual.” [Online]. Available: https://www.gnu.org/software/hurd/gnumach-doc/Task-Information.html

[68] K. B. Wheeler, R. C. Murphy, and D. Thain, “Qthreads: An API for programming with millions of lightweight threads,” in *2008 IEEE International Symposium on Parallel and Distributed Processing*, Apr. 2008. doi: 10.1109/IPDPS.2008.4536359 pp. 1–8, iSSN: 1530-2075. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/4536359

[69] “`DispatchGroup` | Apple Developer Documentation.” [Online]. Available: https://developer.apple.com/documentation/dispatch/dispatchgroup

[70] “`TaskGroup` | Apple Developer Documentation.” [Online]. Available: https://developer.apple.com/documentation/swift/taskgroup

[71] “The Java Grande Forum Multi-threaded Benchmarks - EPCC.” [Online]. Available: https://web.archive.org/web/20140401110543/http://www2.epcc.ed.ac.uk:80/computing/research_activities/java_grande/threads/s1contents.html

[72] "The Innsbruck C++11 Async Benchmark Suite – Github." [Online]. Available: https://github.com/PeterTh/inncabs

[73] "`TaskPriority` | Apple Developer Documentation." [Online]. Available: https://developer.apple.com/documentation/swift/taskpriority

[74] "`os_signpost(_:dso:log:name:signpostID:)` | Apple Developer Documentation." [Online]. Available: https://developer.apple.com/documentation/os/3019241-os_signpost

[75] "`os_proc_available_memory` | Apple Developer Documentation." [Online]. Available: https://developer.apple.com/documentation/os/3191911-os_proc_available_memory

[76] A. Georges, D. Buytaert, and L. Eeckhout, "Statistically rigorous java performance evaluation," *ACM SIGPLAN Notices*, vol. 42, no. 10, pp. 57–76, 2007. doi: 10.1145/1297105.1297033. [Online]. Available: https://doi.org/10.1145/1297105.1297033

[77] "Debugging The Compiler, Printing the Intermediate Representations – Github." [Online]. Available: https://github.com/apple/swift/blob/main/docs/DebuggingTheCompiler.md#printing-the-intermediate-representations

[78] T. Chen, Q. Guo, O. Temam, Y. Wu, Z. Xu, and Y. Chen, "Statistical Performance Comparisons of Computers," *IEEE Transactions on Computers*, 2014. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/6783811

[79] T. Hoefler and R. Belli, "Scientific benchmarking of parallel computing systems: twelve ways to tell the masses when reporting performance results," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. Austin Texas: ACM, Nov. 2015. doi: 10.1145/2807591.2807644. ISBN 978-1-4503-3723-6 pp. 1–12. [Online]. Available: https://dl.acm.org/doi/10.1145/2807591.2807644

[80] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney, "Producing Wrong Data Without Doing Anything Obviously Wrong!" *ACM SIGPLAN Notices*, 2009. [Online]. Available: https://doi.org/10.1145/1508284.1508275

[81] “Goal 9: Industry, innovation, infrastructure | Joint SDG Fund.” [Online]. Available: jointsdgfund.org/sustainable-development-goals/goal-9-industry-innovation-infrastructure

[82] “Goal 12: Responsible consumption, production | Joint SDG Fund.” [Online]. Available: https://www.jointsdgfund.org/sustainable-development-goals/goal-12-responsible-consumption-production

# Appendix A

# Approximate confidence intervals for differences of average execution times

## A.1 SpawnManyWaiting

Table A.1: Difference (GCD - SC) between the means of the execution time results in *SpawnManyWaiting* for N = 2.

| | **SC (-Onone)** | **SC (-O)** | **SC (-Osize)** |
|---|---|---|---|
| **GCD (-Onone)** | 1.180e-03 ± 1.368e-03 | -4.500e-03 ± 9.952e-04 | 2.286e-03 ± 1.343e-03 |
| **GCD (-O)** | 5.665e-04 ± 1.397e-03 | -5.114e-03 ± 1.035e-03 | 1.673e-03 ± 1.372e-03 |
| **GCD (-Osize)** | 6.176e-04 ± 1.410e-03 | -5.062e-03 ± 1.053e-03 | 1.724e-03 ± 1.386e-03 |

Table A.2: Difference (GCD - SC) between the means of the execution time results in *SpawnManyWaiting* for N = 4.

| | **SC (-Onone)** | **SC (-O)** | **SC (-Osize)** |
|---|---|---|---|
| **GCD (-Onone)** | -3.913e-03 ± 2.592e-03 | -6.450e-03 ± 2.525e-03 | -4.703e-03 ± 2.410e-03 |
| **GCD (-O)** | -4.126e-03 ± 2.606e-03 | -6.663e-03 ± 2.539e-03 | -4.916e-03 ± 2.424e-03 |
| **GCD (-Osize)** | -4.332e-03 ± 2.638e-03 | -6.869e-03 ± 2.573e-03 | -5.121e-03 ± 2.460e-03 |

Table A.3: Difference (GCD - SC) between the means of the execution time results in *SpawnManyWaiting* for N = 6.

| | **SC (-Onone)** | **SC (-O)** | **SC (-Osize)** |
|---|---|---|---|
| **GCD (-Onone)** | -1.126e-02 ± 3.473e-03 | -1.235e-02 ± 2.969e-03 | -5.472e-03 ± 3.330e-03 |
| **GCD (-O)** | -1.117e-02 ± 3.482e-03 | -1.226e-02 ± 2.980e-03 | -5.387e-03 ± 3.339e-03 |
| **GCD (-Osize)** | -1.129e-02 ± 3.507e-03 | -1.238e-02 ± 3.008e-03 | -5.498e-03 ± 3.365e-03 |

Table A.4: Difference (GCD - SC) between the means of the execution time results in *SpawnManyWaiting* for N = 8.

| | **SC (-Onone)** | **SC (-O)** | **SC (-Osize)** |
|---|---|---|---|
| **GCD (-Onone)** | -8.542e-03 ± 4.595e-03 | -1.251e-02 ± 4.446e-03 | -6.950e-03 ± 4.138e-03 |
| **GCD (-O)** | -8.845e-03 ± 4.587e-03 | -1.282e-02 ± 4.437e-03 | -7.253e-03 ± 4.129e-03 |
| **GCD (-Osize)** | -8.776e-03 ± 4.580e-03 | -1.275e-02 ± 4.430e-03 | -7.184e-03 ± 4.122e-03 |

Table A.5: Difference (GCD - SC) between the means of the execution time results in *SpawnManyWaiting* for N = 10.

| | **SC (-Onone)** | **SC (-O)** | **SC (-Osize)** |
|---|---|---|---|
| **GCD (-Onone)** | 5.669e-03 ± 5.380e-03 | 1.404e-02 ± 4.125e-03 | 1.274e-02 ± 4.413e-03 |
| **GCD (-O)** | 5.720e-03 ± 5.384e-03 | 1.409e-02 ± 4.130e-03 | 1.279e-02 ± 4.417e-03 |
| **GCD (-Osize)** | 5.272e-03 ± 5.404e-03 | 1.364e-02 ± 4.156e-03 | 1.235e-02 ± 4.442e-03 |

Table A.6: Difference (GCD - SC) between the means of the execution time results in *SpawnManyWaitingGroup* for N = 50.

| | **SC (-Onone)** | **SC (-O)** | **SC (-Osize)** |
|---|---|---|---|
| **GCD (-Onone)** | -7.928e-04 ± 6.780e-04 | -2.758e-03 ± 7.586e-04 | 6.182e-04 ± 8.638e-04 |
| **GCD (-O)** | -8.285e-04 ± 6.770e-04 | -2.794e-03 ± 7.577e-04 | 5.825e-04 ± 8.630e-04 |
| **GCD (-Osize)** | -8.263e-04 ± 6.773e-04 | -2.791e-03 ± 7.580e-04 | 5.847e-04 ± 8.632e-04 |

Table A.7: Difference (GCD - SC) between the means of the execution time results in *SpawnManyWaitingGroup* for N = 100.

| | **SC (-Onone)** | **SC (-O)** | **SC (-Osize)** |
|---|---|---|---|
| **GCD (-Onone)** | 1.025e-01 ± 6.719e-04 | 9.966e-02 ± 4.890e-04 | 1.002e-01 ± 5.419e-04 |
| **GCD (-O)** | 1.025e-01 ± 6.817e-04 | 9.966e-02 ± 5.024e-04 | 1.002e-01 ± 5.540e-04 |
| **GCD (-Osize)** | 1.021e-01 ± 7.080e-04 | 9.921e-02 ± 5.375e-04 | 9.980e-02 ± 5.860e-04 |

Table A.8: Difference (GCD - SC) between the means of the execution time results in *SpawnManyWaitingGroup* for N = 150.

| | **SC (-Onone)** | **SC (-O)** | **SC (-Osize)** |
|---|---|---|---|
| **GCD (-Onone)** | 2.063e-01 ± 6.924e-04 | 2.039e-01 ± 6.194e-04 | 2.043e-01 ± 6.394e-04 |
| **GCD (-O)** | 2.062e-01 ± 6.893e-04 | 2.039e-01 ± 6.159e-04 | 2.042e-01 ± 6.360e-04 |
| **GCD (-Osize)** | 2.056e-01 ± 6.864e-04 | 2.032e-01 ± 6.126e-04 | 2.036e-01 ± 6.328e-04 |

Table A.9: Difference (GCD - SC) between the means of the execution time results in *SpawnManyWaitingGroup* for N = 200.

| | **SC (-Onone)** | **SC (-O)** | **SC (-Osize)** |
|---|---|---|---|
| **GCD (-Onone)** | 3.080e-01 ± 6.854e-04 | 3.059e-01 ± 5.584e-04 | 3.055e-01 ± 4.980e-04 |
| **GCD (-O)** | 3.101e-01 ± 7.465e-04 | 3.081e-01 ± 6.320e-04 | 3.076e-01 ± 5.793e-04 |
| **GCD (-Osize)** | 3.080e-01 ± 7.227e-04 | 3.060e-01 ± 6.037e-04 | 3.055e-01 ± 5.483e-04 |

Table A.10: Difference (GCD - SC) between the means of the execution time results in *SpawnManyWaitingGroup* for N = 250.

| | **SC (-Onone)** | **SC (-O)** | **SC (-Osize)** |
|---|---|---|---|
| **GCD (-Onone)** | 3.082e-01 ± 6.744e-04 | 3.074e-01 ± 6.715e-04 | 3.063e-01 ± 5.411e-04 |
| **GCD (-O)** | 3.091e-01 ± 6.747e-04 | 3.083e-01 ± 6.718e-04 | 3.072e-01 ± 5.414e-04 |
| **GCD (-Osize)** | 3.089e-01 ± 6.618e-04 | 3.080e-01 ± 6.588e-04 | 3.069e-01 ± 5.253e-04 |

Table A.11: Difference (GCD - SC) between the means of the execution time results in *SpawnManyWaitingGroup* for N = 300.

| | **SC (-Onone)** | **SC (-O)** | **SC (-Osize)** |
|---|---|---|---|
| **GCD (-Onone)** | 4.135e-01 ± 7.406e-04 | 4.128e-01 ± 6.796e-04 | 4.121e-01 ± 5.824e-04 |
| **GCD (-O)** | 4.132e-01 ± 7.604e-04 | 4.125e-01 ± 7.012e-04 | 4.119e-01 ± 6.075e-04 |
| **GCD (-Osize)** | 4.132e-01 ± 7.279e-04 | 4.125e-01 ± 6.658e-04 | 4.119e-01 ± 5.662e-04 |

## A.2 SpawnManyWaitingGroup

## A.3 SpawnManyActors

Table A.12: Difference (GCD - SC) between the means of the execution time results in *SpawnManyActors* for N = 100.

| | **SC (-Onone)** | **SC (-O)** | **SC (-Osize)** |
|---|---|---|---|
| **GCD (-Onone)** | 1.442e-04 ± 1.518e-05 | 2.316e-04 ± 1.360e-05 | 2.357e-04 ± 1.352e-05 |
| **GCD (-O)** | 1.009e-04 ± 1.372e-05 | 1.883e-04 ± 1.194e-05 | 1.923e-04 ± 1.185e-05 |
| **GCD (-Osize)** | 1.048e-04 ± 1.366e-05 | 1.921e-04 ± 1.188e-05 | 1.962e-04 ± 1.179e-05 |

Table A.13: Difference (GCD - SC) between the means of the execution time results in *SpawnManyActors* for N = 200.

| | **SC (-Onone)** | **SC (-O)** | **SC (-Osize)** |
|---|---|---|---|
| **GCD (-Onone)** | 1.286e-04 ± 9.280e-06 | 2.326e-04 ± 8.338e-06 | 2.417e-04 ± 7.563e-06 |
| **GCD (-O)** | 8.984e-05 ± 9.264e-06 | 1.938e-04 ± 8.319e-06 | 2.029e-04 ± 7.543e-06 |
| **GCD (-Osize)** | 7.957e-05 ± 1.028e-05 | 1.836e-04 ± 9.436e-06 | 1.927e-04 ± 8.759e-06 |

Table A.14: Difference (GCD - SC) between the means of the execution time results in *SpawnManyActors* for N = 300.

| | **SC (-Onone)** | **SC (-O)** | **SC (-Osize)** |
|---|---|---|---|
| **GCD (-Onone)** | 1.415e-04 ± 4.406e-06 | 2.031e-04 ± 4.554e-06 | 2.151e-04 ± 4.163e-06 |
| **GCD (-O)** | 7.273e-05 ± 5.660e-06 | 1.343e-04 ± 5.777e-06 | 1.463e-04 ± 5.474e-06 |
| **GCD (-Osize)** | 8.259e-05 ± 5.988e-06 | 1.442e-04 ± 6.098e-06 | 1.562e-04 ± 5.812e-06 |

Table A.15: Difference (GCD - SC) between the means of the execution time results in *SpawnManyActors* for N = 400.

| | **SC (-Onone)** | **SC (-O)** | **SC (-Osize)** |
|---|---|---|---|
| **GCD (-Onone)** | 2.354e-04 ± 4.277e-06 | 2.918e-04 ± 5.376e-06 | 3.087e-04 ± 3.006e-06 |
| **GCD (-O)** | 1.167e-04 ± 4.129e-06 | 1.731e-04 ± 5.258e-06 | 1.900e-04 ± 2.790e-06 |
| **GCD (-Osize)** | 1.291e-04 ± 4.270e-06 | 1.855e-04 ± 5.370e-06 | 2.024e-04 ± 2.996e-06 |

Table A.16: Difference (GCD - SC) between the means of the execution time results in *SpawnManyActors* for N = 500.

| | **SC (-Onone)** | **SC (-O)** | **SC (-Osize)** |
|---|---|---|---|
| **GCD (-Onone)** | 3.333e-04 ± 3.168e-06 | 4.018e-04 ± 4.814e-06 | 4.164e-04 ± 3.007e-06 |
| **GCD (-O)** | 1.775e-04 ± 2.818e-06 | 2.460e-04 ± 4.591e-06 | 2.606e-04 ± 2.635e-06 |
| **GCD (-Osize)** | 1.944e-04 ± 3.068e-06 | 2.629e-04 ± 4.749e-06 | 2.775e-04 ± 2.901e-06 |

Table A.17: Difference (GCD - SC) between the means of the execution time results in *SpawnManyActors* for N = 600.

| | **SC (-Onone)** | **SC (-O)** | **SC (-Osize)** |
|---|---|---|---|
| **GCD (-Onone)** | 3.934e-04 ± 3.595e-06 | 5.062e-04 ± 4.269e-06 | 5.004e-04 ± 4.818e-06 |
| **GCD (-O)** | 2.045e-04 ± 3.169e-06 | 3.173e-04 ± 3.917e-06 | 3.116e-04 ± 4.509e-06 |
| **GCD (-Osize)** | 2.302e-04 ± 3.788e-06 | 3.430e-04 ± 4.433e-06 | 3.373e-04 ± 4.964e-06 |

Table A.18: Difference (GCD - SC) between the means of the execution time results in *SpawnManyActors* for N = 700.

| | **SC (-Onone)** | **SC (-O)** | **SC (-Osize)** |
|---|---|---|---|
| **GCD (-Onone)** | 4.620e-04 ± 3.882e-06 | 6.117e-04 ± 3.894e-06 | 6.091e-04 ± 4.208e-06 |
| **GCD (-O)** | 2.428e-04 ± 3.344e-06 | 3.925e-04 ± 3.358e-06 | 3.899e-04 ± 3.718e-06 |
| **GCD (-Osize)** | 2.738e-04 ± 5.451e-06 | 4.235e-04 ± 5.459e-06 | 4.209e-04 ± 5.688e-06 |

Table A.19: Difference (GCD - SC) between the means of the execution time results in *SpawnManyActors* for N = 800.

| | **SC (-Onone)** | **SC (-O)** | **SC (-Osize)** |
|---|---|---|---|
| **GCD (-Onone)** | 5.046e-04 ± 3.230e-06 | 6.770e-04 ± 4.242e-06 | 6.830e-04 ± 3.811e-06 |
| **GCD (-O)** | 2.821e-04 ± 2.992e-06 | 4.545e-04 ± 4.064e-06 | 4.606e-04 ± 3.612e-06 |
| **GCD (-Osize)** | 2.988e-04 ± 3.753e-06 | 4.712e-04 ± 4.653e-06 | 4.773e-04 ± 4.263e-06 |

Table A.20: Difference (GCD - SC) between the means of the execution time results in *SpawnManyActors* for N = 900.

| | **SC (-Onone)** | **SC (-O)** | **SC (-Osize)** |
|---|---|---|---|
| **GCD (-Onone)** | 5.573e-04 ± 2.803e-06 | 7.540e-04 ± 4.019e-06 | 7.582e-04 ± 4.024e-06 |
| **GCD (-O)** | 3.155e-04 ± 3.269e-06 | 5.122e-04 ± 4.356e-06 | 5.163e-04 ± 4.362e-06 |
| **GCD (-Osize)** | 3.194e-04 ± 2.605e-06 | 5.161e-04 ± 3.883e-06 | 5.202e-04 ± 3.889e-06 |

Table A.21: Difference (GCD - SC) between the means of the execution time results in *SpawnManyActors* for N = 1000.

| | **SC (-Onone)** | **SC (-O)** | **SC (-Osize)** |
|---|---|---|---|
| **GCD (-Onone)** | 6.180e-04 ± 2.775e-06 | 8.341e-04 ± 3.020e-06 | 8.378e-04 ± 3.506e-06 |
| **GCD (-O)** | 3.507e-04 ± 3.267e-06 | 5.668e-04 ± 3.478e-06 | 5.705e-04 ± 3.907e-06 |
| **GCD (-Osize)** | 3.583e-04 ± 2.668e-06 | 5.744e-04 ± 2.922e-06 | 5.781e-04 ± 3.422e-06 |

## A.4 Fibonacci

Table A.22: Difference (GCD - SC) between the means of the execution time results in *Fibonacci* for N = 5.

| | **SC (-Onone)** | **SC (-O)** | **SC (-Osize)** |
|---|---|---|---|
| **GCD (-Onone)** | 1.103e-05 ± 5.230e-06 | 2.175e-05 ± 4.622e-06 | 2.306e-05 ± 4.615e-06 |
| **GCD (-O)** | 1.454e-05 ± 7.652e-06 | 2.525e-05 ± 7.251e-06 | 2.656e-05 ± 7.246e-06 |
| **GCD (-Osize)** | 1.109e-05 ± 5.842e-06 | 2.180e-05 ± 5.305e-06 | 2.312e-05 ± 5.299e-06 |

Table A.23: Difference (GCD - SC) between the means of the execution time results in *Fibonacci* for N = 10.

| | **SC (-Onone)** | **SC (-O)** | **SC (-Osize)** |
|---|---|---|---|
| **GCD (-Onone)** | 2.309e-05 ± 2.799e-05 | 8.839e-05 ± 2.673e-05 | 8.825e-05 ± 2.708e-05 |
| **GCD (-O)** | 8.918e-06 ± 2.650e-05 | 7.421e-05 ± 2.518e-05 | 7.407e-05 ± 2.554e-05 |
| **GCD (-Osize)** | 1.269e-05 ± 2.561e-05 | 7.798e-05 ± 2.424e-05 | 7.785e-05 ± 2.461e-05 |

Table A.24: Difference (GCD - SC) between the means of the execution time results in *Fibonacci* for N = 15.

| | **SC (-Onone)** | **SC (-O)** | **SC (-Osize)** |
|---|---|---|---|
| **GCD (-Onone)** | -1.760e-04 ± 3.962e-05 | 6.996e-05 ± 3.973e-05 | 3.821e-05 ± 4.181e-05 |
| **GCD (-O)** | -2.336e-04 ± 4.114e-05 | 1.227e-05 ± 4.125e-05 | -1.947e-05 ± 4.326e-05 |
| **GCD (-Osize)** | -2.389e-04 ± 4.066e-05 | 7.010e-06 ± 4.077e-05 | -2.474e-05 ± 4.280e-05 |

Table A.25: Difference (GCD - SC) between the means of the execution time results in *Fibonacci* for N = 20.

| | **SC (-Onone)** | **SC (-O)** | **SC (-Osize)** |
|---|---|---|---|
| **GCD (-Onone)** | -4.608e-03 ± 1.374e-04 | 2.783e-04 ± 8.373e-05 | 4.068e-04 ± 8.334e-05 |
| **GCD (-O)** | -5.246e-03 ± 1.380e-04 | -3.603e-04 ± 8.465e-05 | -2.318e-04 ± 8.427e-05 |
| **GCD (-Osize)** | -5.320e-03 ± 1.372e-04 | -4.343e-04 ± 8.331e-05 | -3.058e-04 ± 8.292e-05 |

Table A.26: Difference (GCD - SC) between the means of the execution time results in *Fibonacci* for N = 25.

| | **SC (-Onone)** | **SC (-O)** | **SC (-Osize)** |
|---|---|---|---|
| **GCD (-Onone)** | -1.293e-01 ± 6.152e-04 | -4.470e-02 ± 5.859e-04 | -4.497e-02 ± 6.063e-04 |
| **GCD (-O)** | -1.375e-01 ± 6.176e-04 | -5.293e-02 ± 5.884e-04 | -5.319e-02 ± 6.087e-04 |
| **GCD (-Osize)** | -1.358e-01 ± 6.370e-04 | -5.122e-02 ± 6.088e-04 | -5.148e-02 ± 6.285e-04 |

Table A.27: Difference (GCD - SC) between the means of the execution time results in *NQueens* for N = 1.

| | **SC (-Onone)** | **SC (-O)** | **SC (-Osize)** |
|---|---|---|---|
| **GCD (-Onone)** | 4.997e-05 ± 6.056e-06 | 6.634e-05 ± 4.276e-06 | 6.477e-05 ± 4.309e-06 |
| **GCD (-O)** | 4.588e-05 ± 6.565e-06 | 6.225e-05 ± 4.971e-06 | 6.068e-05 ± 4.999e-06 |
| **GCD (-Osize)** | 4.211e-05 ± 6.722e-06 | 5.848e-05 ± 5.176e-06 | 5.691e-05 ± 5.204e-06 |

Table A.28: Difference (GCD - SC) between the means of the execution time results in *NQueens* for N = 2.

| | **SC (-Onone)** | **SC (-O)** | **SC (-Osize)** |
|---|---|---|---|
| **GCD (-Onone)** | 3.263e-05 ± 4.562e-06 | 5.662e-05 ± 4.451e-06 | 5.776e-05 ± 4.320e-06 |
| **GCD (-O)** | 1.455e-05 ± 4.632e-06 | 3.853e-05 ± 4.523e-06 | 3.967e-05 ± 4.394e-06 |
| **GCD (-Osize)** | 1.442e-05 ± 4.118e-06 | 3.841e-05 ± 3.995e-06 | 3.955e-05 ± 3.849e-06 |

Table A.29: Difference (GCD - SC) between the means of the execution time results in *NQueens* for N = 3.

| | **SC (-Onone)** | **SC (-O)** | **SC (-Osize)** |
|---|---|---|---|
| **GCD (-Onone)** | 7.622e-05 ± 1.096e-05 | 1.180e-04 ± 1.071e-05 | 1.177e-04 ± 1.069e-05 |
| **GCD (-O)** | 4.298e-05 ± 7.702e-06 | 8.479e-05 ± 7.342e-06 | 8.443e-05 ± 7.306e-06 |
| **GCD (-Osize)** | 4.703e-05 ± 6.791e-06 | 8.884e-05 ± 6.379e-06 | 8.847e-05 ± 6.338e-06 |

Table A.30: Difference (GCD - SC) between the means of the execution time results in *NQueens* for N = 4.

| | **SC (-Onone)** | **SC (-O)** | **SC (-Osize)** |
|---|---|---|---|
| **GCD (-Onone)** | 1.961e-04 ± 2.071e-05 | 2.847e-04 ± 2.033e-05 | 2.885e-04 ± 2.042e-05 |
| **GCD (-O)** | 1.510e-04 ± 1.404e-05 | 2.396e-04 ± 1.347e-05 | 2.435e-04 ± 1.360e-05 |
| **GCD (-Osize)** | 1.485e-04 ± 1.207e-05 | 2.371e-04 ± 1.140e-05 | 2.410e-04 ± 1.156e-05 |

Table A.31: Difference (GCD - SC) between the means of the execution time results in *NQueens* for N = 5.

| | **SC (-Onone)** | **SC (-O)** | **SC (-Osize)** |
|---|---|---|---|
| **GCD (-Onone)** | 3.855e-03 ± 2.104e-03 | 3.981e-03 ± 2.104e-03 | 3.990e-03 ± 2.104e-03 |
| **GCD (-O)** | 1.905e-03 ± 1.452e-03 | 2.030e-03 ± 1.452e-03 | 2.039e-03 ± 1.452e-03 |
| **GCD (-Osize)** | 2.352e-03 ± 1.719e-03 | 2.478e-03 ± 1.719e-03 | 2.487e-03 ± 1.719e-03 |

## A.5 NQueens

## A.6 MatrixMultiplication

Table A.32: Difference (GCD - SC) between the means of the execution time results in *MatrixMultiplication* for N = 25.

| | **SC (-Onone)** | **SC (-O)** | **SC (-Osize)** |
|---|---|---|---|
| **GCD (-Onone)** | 5.044e-03 ± 8.831e-05 | 7.276e-03 ± 1.187e-04 | 7.297e-03 ± 1.101e-04 |
| **GCD (-O)** | -9.945e-04 ± 5.282e-05 | 1.237e-03 ± 9.533e-05 | 1.258e-03 ± 8.438e-05 |
| **GCD (-Osize)** | -9.464e-04 ± 5.244e-05 | 1.285e-03 ± 9.512e-05 | 1.306e-03 ± 8.414e-05 |

Table A.33: Difference (GCD - SC) between the means of the execution time results in *MatrixMultiplication* for N = 50.

| | **SC (-Onone)** | **SC (-O)** | **SC (-Osize)** |
|---|---|---|---|
| **GCD (-Onone)** | 3.325e-02 ± 3.931e-04 | 4.751e-02 ± 3.734e-04 | 4.740e-02 ± 3.831e-04 |
| **GCD (-O)** | -2.824e-03 ± 3.311e-04 | 1.143e-02 ± 3.074e-04 | 1.133e-02 ± 3.192e-04 |
| **GCD (-Osize)** | -2.619e-03 ± 3.252e-04 | 1.164e-02 ± 3.010e-04 | 1.153e-02 ± 3.131e-04 |

Table A.34: Difference (GCD - SC) between the means of the execution time results in *MatrixMultiplication* for N = 75.

| | **SC (-Onone)** | **SC (-O)** | **SC (-Osize)** |
|---|---|---|---|
| **GCD (-Onone)** | 9.985e-02 ± 1.352e-03 | 1.263e-01 ± 6.246e-04 | 1.262e-01 ± 6.516e-04 |
| **GCD (-O)** | -1.070e-02 ± 1.988e-03 | 1.577e-02 ± 1.586e-03 | 1.565e-02 ± 1.597e-03 |
| **GCD (-Osize)** | -1.187e-02 ± 1.751e-03 | 1.460e-02 ± 1.276e-03 | 1.448e-02 ± 1.289e-03 |

Table A.35: Difference (GCD - SC) between the means of the execution time results in *MatrixMultiplication* for N = 100.

| | **SC (-Onone)** | **SC (-O)** | **SC (-Osize)** |
|---|---|---|---|
| **GCD (-Onone)** | 1.985e-01 ± 8.932e-03 | 2.835e-01 ± 8.562e-03 | 2.830e-01 ± 8.559e-03 |
| **GCD (-O)** | -6.801e-02 ± 4.631e-03 | 1.695e-02 ± 3.871e-03 | 1.642e-02 ± 3.864e-03 |
| **GCD (-Osize)** | -6.579e-02 ± 4.502e-03 | 1.917e-02 ± 3.715e-03 | 1.864e-02 ± 3.708e-03 |

TRITA-EECS-EX-2022:713