Safia Guellil

A Comprehensive Review of Golang to Web Assembly Conversion with TinyGo

Unlocking the Power of Web Assembly for Golang

Final Master's Project

Directed by Dr. Marc Sánchez Artigas

University Master's Degree in Computer Security Engineering and Artificial Intelligence

UNIVERSITAT ROVIRA I VIRGILI

Tarragona
2023

Abstract

WebAssembly is a bytecode language that has revolutionized web browsing by enabling high-performance code execution. Its outstanding qualities, such as portability, scalability, security, and speed, combined with the emergence of the WebAssembly System Interface standard, are bringing the vision of "*write once, run anywhere*" closer to reality. As a result, its influence has spread beyond web browsers, leading many programming languages, including Golang, to compile code for execution outside the browser. This project examines the conversion of WebAssembly to Golang using the TinyGo compiler for non-Web environments. We evaluate the memory system, which is fundamental in Golang, and the performance of the WebAssembly module compiled with TinyGo. In the analysis performed, we observed that the Golang memory system is reproduced in the WASM module but in a simplified version of TinyGo. In addition, we achieved execution speeds comparable to the Golang source language but with a significantly reduced binary size, which makes TinyGo and Golang a good choice for resource-constrained environments.

Keywords: *WebAssembly*, *Golang*, *TinyGo*

List Of Figures

Figure 1. Web Assembly logo. https://www.w3.org/TR/wasm-core-1/ 13
Figure 2. WebAssembly Goals. https://geekflare.com/webassembly-wasm-goals-key-concepts-use-cases 15
Figure 3. Indirect function calls via the table section. https://www.usenix.org/conference/usenixsecurity20/presentation/lehmann 16
Figure 4. Representation of the code in C, .wat and .wasm formats. https://en.wikipedia.org/wiki/WebAssembly 17
Figure 5. WASI icon. https://blog.vasquezruiz.me/wasi-una-interfaz-de-sistemas-para-webassembly/ 18
Figure 6. Google Earth's logo. https://es.wikipedia.org/wiki/Archivo:Logotipo_de_Google_Earth.png 20
Figure 7. Shopify Wasm Engine. https://shopify.engineering/shopify-webassembly 20
Figure 8. Golang icon. https://hacks.mozilla.org/2019/03/standardizing-wasi-a-webassembly-system-interface/ 21
Figure 9. Docker icon. https://www.docker.com/company/newsroom/media-resources/ 22
Figure 10. Kubernetes icon. https://kubernetes.io/ 22
Figure 11. TinyGo's logo. https://tinygo.org 24
Figure 12. Conversion from Golang to WebAssembly. Figure modified from: https://www.alibabacloud.com/blog/using-webassembly-and-kubernetes-in-combination_596177 26
Figure 13. Basic Memory Layout. https://study.com/academy/lesson/how-to-allocate-deallocate-memory-in-c-programming.html 27
Figure 14. Heap allocation and deallocation. https://medium.com/safetycultureengineering/an-overview-of-memory-management-in-go-9a72ec7c76a8 28
Figure 15. Stack allocation and deallocation. https://medium.com/safetycultureengineering/an-overview-of-memory-management-in-go-9a72ec7c76a8 28
Figure 16. Go Program's Memory. https://dev.to/karankumarshreds/memory-allocations-in-go-1bpa 29
Figure 17. Escape to the heap behavior. Generated by author Safia Guellil. 31
Figure 18. Example of escape to the heap behavior .Generated by author Safia Guellil. 32
Figure 19. Example of escape to the heap behavior. Generated by author Safia Guellil. 32
Figure 20. Go's Garbage Collector process. https://en.wikipedia.org/wiki/Tracing_garbage_collection 34
Figure 21. Garbage Collection Triggers. Generated by author Safia Guellil. 35
Figure 22. TinyGo Memory Layout. https://aykevl.nl/2020/09/gc-tinygo 36
Figure 24. Escape to the heap in TinyGo. https://tinygo.org/docs/concepts/compiler-internals/heap-allocation/ 39
Figure 24. Example where the pointer doesn't escape to the heap on TinyGo. https://tinygo.org/docs/concepts/compiler-internals/heap-allocation/ 39
Figure 25. WASM linear memory for different compilers. https://www.usenix.org/system/files/sec20-lehmann.pdf 40
Figure 26. WASM linear memory for different compilers. https://www.usenix.org/system/files/sec20-lehmann.pdf 48
Figure 27. WASM Linear Memory from TinyGo. Generated by the author Safia Guellil. 48
Figure 28. 100 execution time of Escape code. Generated by the author Safia Guellil. 50
Figure 29. 100 execution time of non-Escape code. Generated by the author Safia Guellil. 50
Figure 30. 100 execution time of Escape code (allocating large object on stack). Generated by the author Safia Guellil. 50
Figure 31. Call Graph of the execution on a wasm module compiled with TinyGo. Generated by the author Safia Guellil. 53
Figure 32. Number of Garbage Collector cycles. Generated by the author Safia Guellil. 54
Figure 33. Number of freed objects. Generated by the author Safia Guellil. 55

Figure 34. Graph interface of Floyd-Warshall algorithm. https://rosettacode.org/wiki/Floyd-Warshall_algorithm 59
Figure 35. Struct in KNN algorithm. https://github.com/mattn/go-knn-iris/tree/master 60
Figure 36. Struct to show results. https://medium.com/@snassr/processing-large-files-in-go-golang-6ea87effbfe2 61
Figure 37. File with employee information. https://medium.com/@snassr/processing-large-files-in-go-golang-6ea87effbfe2 61
Figure 38. Parallel matrix multiplication. Generated by the author Safia Guellil. 61
Figure 39. Wasmedge runtime performance. Generated by the author Safia Guellil. 66
Figure 40. Wasmtime and Wazero performance on File System and FASTA algorithms. Generated by the author Safia Guellil. 66
Figure 41. Runtimes performance with the File System (LARGE) algorithm. Generated by Safia Guellil. 67
Figure 42. Average of 100 runs with different runtimes and optimization levels. Generated by the author Safia Guellil. 68
Figure 43. Performance with and without conservative garbage collector. Generated by the author Safia Guellil. 69
Figure 44. Number of System Calls for each benchmark and runtime. Generated by the author Safia Guellil. 72
Figure 45. Parallel Matrix Multiplication algorithm comparison with Golang and TinyGo. Generated by the author Safia Guellil. 78

List Of Tables

Table 1. Comparison between WebAssembly compiler. https://doi.org/10.3390/fi15080275.....12
Table 2. Comparison between frameworks for WebAssembly. https://doi.org/10.3390/fi15080275 ..12
Table 3. Comparison between JavaScript and WebAssembly. https://snipcart.com/blog/webassembly-vs-javascript.....14
Table 4. System environment. Generated by the author Safia Guellil.....45
Table 5. Comparison of binary sizes. Generated by the author Safia Guellil.....55
Table 6. Comparison of execution times. Generated by the author Safia Guellil.....55
Table 7. Execution time of one GC cycle. Generated by the author Safia Guellil. 56
Table 8. Comparison of binary sizes. Generated by the author Safia Guellil.....56
Table 9. Suite of Benchmarks. Generated by the author Safia Guellil. 58
Table 10. Selected Runtimes. Generated by the author Safia Guellil.....62
Table 11. Discarded Runtimes. Generated by the author Safia Guellil. 63
Table 12. TinyGo interesting building flags. Generated by the author Safia Guellil. 64
Table 13. Best execution times. Generated by the author Safia Guellil. 65
Table 14. Binary sizes for each optimization level of TinyGo. Generated by the author Safia Guellil. 70
Table 15. Best number of system calls (independently of the optimization level). Generated by the author Safia Guellil.....71
Table 16. System Calls of each runtime with the Quicksort algorithm. Generated by the author Safia Guellil. 73
Table 17. Jaccard Similarity Coefficient. Generated by the author Safia Guellil.....74
Table 18. Sørensen-Dice Coefficient. Generated by the author Safia Guellil. 75
Table 19. Output of the matrix multiplication algorithm with Golang. Generated by the author Safia Guellil. 77
Table 20. Output of the matrix multiplication algorithm with TinyGo. Generated by the author Safia Guellil. 78

List Of Abbreviations

WASM: Web Assembly
WASI: Web Assembly Interface
STW: Stop the World
MVP: Minimum Viable Product
W3C: World Wide Web Consortium
LIFO: Last In First Out
GC: Garbage Collector

Index

1. Introduction *8*

1.1. Objectives **10**

1.2. State of the Art **11**

2. Background *13*

2.1. Overview of WebAssembly **13**
2.1.1. Brief introduction 13
2.1.2. WebAssembly vs JavaScript 14
2.1.3. General Features 15
2.1.4. Web Assembly Goals 15
2.1.5. Web Assembly Specification 16
2.1.6. Web Assembly Interface 18
2.1.7. Web Assembly Runtimes 19
2.1.9. Web Assembly Real cases 20

2.2. Overview of Golang **21**
2.2.1. Strengths of Go compared to C 21
2.2.2. Real-world applications implemented with Go 22
2.2.3. Compile Golang to WASM 23

2.3. Overview of TinyGo **24**
2.3.1. TinyGo Goals 24
2.3.2. Compiler Advantages and Disadvantages 25

2.4. Golang to WebAssembly conversion **26**

3. Memory management *27*

3.1. Basic concepts of memory management **27**
3.1.1. Memory Layout 27
3.1.2. Memory Allocation 28
3.1.3. Memory Deallocation 28

3.2. Memory management in Golang **29**
3.2.1. Goroutines memory allocation 29
3.2.2. Golang's memory escape behavior 30
3.2.3. Go's Garbage Collector 33

3.3. Memory management in TinyGo **36**
3.3.1. Memory Layout 36
3.3.2. Memory management strategies 36
3.3.3. TinyGo's Garbage Collector 37
3.3.4. TinyGo's Heap Allocation 39

3.4. Memory management in Web Assembly **40**
3.4.1. Managed and Unmanaged Data 40
3.4.2. Linear Memory Layout 40
3.4.3. WASM's Garbage Collector Proposal 41

3.5. Conclusion about Memory Management **42**
3.5.1. Memory Layout 42
3.5.2. Escape to the heap 42
3.5.3. Garbage collector 42

4. WebAssembly Evaluation *43*

4.1. Evaluation questions and methodology **43**
4.1.1. Methodology for assessing memory management 43
4.1.2. Methodology for assessing performance 44

4.1.3. Evaluation Environment 45

4.2. Evaluation of Memory Management **46**

4.2.1. Q1: Analyzing the linear memory layout structure 46

4.2.2. Q2: Analyzing the Escape to the heap behavior 49

4.2.3. Q3: Analyzing the Garbage Collector 51

4.2.4. Conclusions of the memory management evaluation 57

4.3. Evaluation of Performance **58**

4.3.1. Suite of Benchmarks 58

4.3.2. Suite of Runtimes 62

4.3.3. Building Flags 64

4.3.4. Q1: Execution Time 65

4.3.5. Q2: Binary size 70

4.3.6. Q3: System Calls 71

4.3.7. Conclusions of the performance evaluation 76

4.3.8. Special Benchmarks 77

5. Conclusions ***80***

5.1. Future Work **81**

References ***82***

1. Introduction

WebAssembly (abbreviated as WASM) is a bytecode language that emerged in 2015 as a promising project that would improve performance on web pages [51]. Until then, the only language that could run on the web was JavaScript which was not ready to meet the growing demand of browsers. However, since the implementation of this project in 2017, WebAssembly has become a widely adopted technology that has quickly gained a lot of popularity. This has been such that in 2019, it was recognized as an official language by the World Wide Web Consortium (W3C) being the fourth language after HTML, CSS, and JavaScript that can run on browsers [52].

This popularity stems mainly from its portability, security, scalability, and speed [38], which has benefited large projects such as Google Earth that have used WebAssembly to expand to more browsers. In fact, the project started in 2017 with the support of just four browsers (Chrome, Edge, Firefox, and WebKit) [8]. However, as of April, 96% of browsers can run wasm code [51]. These examples show how important and impactful WebAssembly is on the web, but it doesn't end there; thanks to its remarkable features such as speed, scalability, and security, users started to use WASM code outside the web [9].

So far, Web Assembly has been able to run inside the browser thanks to the JavaScript engine. However, outside the web, it needed an interface that facilitates interaction with the operating system to use its resources.

"*WebAssembly is an assembly language for a conceptual machine, not a physical one*." [9]. This means that it is not defined for a specific architecture, which allows it to run on a wide variety of machines. But it also means that the interface it needs must be for a conceptual operating system, not for a single operating system [9]. This is where the WASI or WebAssembly Interface project comes in in 2019. WASI is a standard API that allows access to different operating system features such as the file system, clocks, or random numbers [4].

With the emergence of this new technology, the full potential of WASM code portability has been realized. Both WebAssembly itself and the novel WASI interface can run on all architectures and operating systems. In fact, Docker co-founder Solomon Hykes tweeted in 2019 [39]:

> *how important it is. WebAssembly on the server is the future of computing.”.*
>
> ***Solomon Hykes, co-founder of Docker***

Now that WebAssembly can run both inside and outside the browser, many programming languages and compilers, started to adapt and support this technology. So that "*write once, run anywhere*" [2] dream seems to be taking shape with the rise of WebAssembly.

However, it still has a long way to go as WASI is still in preview1 and Web Assembly still has many proposals to supply. Among them, one proposal stands out that, since 2017 (first implementation), has been a clear goal: the garbage collector, an automatic memory manager [8]. So far, Web Assembly does not have a memory management system, since supporting one means adding new tools. Therefore, languages such as Golang that excel in memory management systems must compile their garbage collectors, which entails an additional cost both in performance and binary size [17]. Nevertheless, this small problem can be cushioned with the compiler TinyGo.

TinyGo is a project designed to be able to run Go code on all small microcontrollers, mainly by reducing the size of the binary, which is consequently faster. This project has most of the features of Golang, such as the garbage collector, but in many cases is simpler [1]. On the other hand, it also allows compiling the GO code to WebAssembly both inside and outside the browser with the WASI interface. This is quite advantageous, since reducing the size of the wasm module allows it to be faster and more portable.

As TinyGo is a little-researched tool, especially in the WebAssembly sector outside the browser, we thought it would be interesting to study it and contribute something new to the scientific community. In this project, we analyze how this tool compiles GoLang code to WebAssembly, with a special focus on the memory management system so prominent in the Go language.

1.1. Objectives

As mentioned in the introduction, the compilation of Golang to WebAssembly via TinyGo and for out-of-browser environments is an under-researched sector as it is a recent development. In this project, we will focus on analyzing the WASM conversion, mainly focusing on the system memory management and performance.

The main objectives of this project are:

1. Learn: how to program in Golang language and its key distinguishing features; the fundamentals of WebAssembly, which include the organization of the wasm module and its human-readable version (.wat); and finally, the basics of the TinyGo compiler.

2. Analyze one of the major features of Golang, the automatic memory management system. Also, examine the internals of memory management in TinyGo and Web Assembly to verify if the memory management system of the source language, Golang, is preserved in the compiled. wasm module.
   - First, we will analyze the features of the Golang memory management system.
   - Then, we will investigate whether these features are present in the TinyGo compiler.
   - Finally, we will check if these features are inherited in the .wasm module.

3. Analyze the different optimization options and the general building flags of TinyGo, and how they improve performance and size.

4. Implement a small benchmarking suite with multiple applications written in Golang and compile them to WebAssembly with TinyGo.

5. Select a set of runtimes to execute the compiled .wasm code and compare the obtained results, according to the optimization flags, binary sizes and system calls.

1.2. State of the Art

Since the emergence of WebAssembly as a language for running high-performance applications in browsers, many articles have been published due to its unique features of speed, portability, and security.

An example would be "*Understanding the performance of WebAssembly applications*", an article aimed at analyzing the performance of WebAssembly in comparison with the commonly used JavaScript [53]. Among their findings, they observed that compilers built on LLVM have optimizations that are not designed for WebAssembly; the performance of the .wasm module varies depending on the execution environment; and that WebAssembly uses more memory than its JavaScript counterparts. This research is based on C-to-WebAssembly compilers [53], so, in terms of Golang, the results could vary.

Another example is "*Everything Old is New Again: Binary Security of WebAssembly*" [31]. This article demonstrates that vulnerabilities in source languages can translate into vulnerabilities in WebAssembly binaries. For instance, WASM allows attacks such as overwriting supposedly constant data or manipulating the heap via an overflow [31]. In addition to these findings, the authors discuss some protection mechanisms to mitigate these risks.

In terms of portability, we have articles such as "*WebAssembly Orchestration in the Context of Serverless Computing*" [29], which present a way to extend Kubernetes by orchestrating natively executed WebAssembly modules. The evaluation results (obtained using WASM modules and OpenFaaS functions) show that WebAssembly has advantages for frequently executed serverless functions requiring elasticity, as it is twice as fast, has a smaller size, and has comparable performance. However, when it comes to sustained performance, containers are a better choice [29].

As for Golang, there are several blogs, forums, web pages, and even books that describe how to compile Go's code to WebAssembly and run it together with JavaScript on browsers. For instance, in the book "*Mastering Go*" by Mihalis Tsoukalos, besides explaining the internal functionalities of the Go language, he shows explicit examples of how to compile and execute a .wasm module in a browser. However, we have not found any articles that specifically analyze the conversion or performance of compiling Golang to Web Assembly.

Regarding Tinygo, we can find articles that evaluate its efficiency on microcontrollers, such as "*Performance Evaluation of C/C++, MicroPython, Rust and TinyGo Programming Languages on ESP32 Microcontroller*" [33], where they recommend TinyGo as efficient higher-level ESP32 programming languages which is fast compared to C and C++.

On the other hand, in relation to WebAssembly, we can find the following book: "*Creative DIY Microcontroller Projects with TinyGo and WebAssembly*" [13]. This book shows several examples where TinyGo is used in modern browsers to display application statistics embedded in WebAssembly panels. However, there are no examples outside the browser.

Finally, the following article links the three main concepts of this thesis (Web Assembly, Golang and Tinygo) outside of browsers: "*An Overview of WebAssembly for IoT: Background, Tools, State-of-the-Art, Challenges, and Future Directions*". This article explores the relationship between IoT technology and WebAssembly and provides a comparison between compilers, runtimes, and source languages (where Golang and TinyGo are also included). As we can see in Table 1, TinyGo has high performance, IoT support and low complexity. On the other hand, in Table 2, we see that Golang has high performance too, high build optimizations, good browser compatibility, and a low learning curve [34].

However, despite containing the main elements, this article is a state of the art of Web assembly for IoT technologies. Therefore, it does not provide performance tests or in-depth analysis regarding the conversion from Golang to WebAssembly with TinyGo.

In conclusion, nowadays, there are more and more articles related to WebAssembly to analyze its different features. However, at the moment, there is no analysis of the compilation of Golang to WebAssembly related to performance, advantages and disadvantages of using the TinyGo compiler and its different optimizations, and above all, focused on non-browser environments where the WASM module is executed through runtimes.

This project aims to provide this small contribution, where we will analyze the memory management system of Golang and see if its features are transferred to TinyGo. In addition, we will run different tests to analyze the performance with different levels of optimization.

Table 1. Comparison between WebAssembly compiler. *https://doi.org/10.3390/fi15080275*

Compiler	Language Support	Optimization	Community Support	Ecosystem	Performance	IoT Support	Development Complexity	Documentation
Emscripten	C/C++	High	High	Wide	Moderate	Yes	Moderate	Extensive
tinygo	Go	Moderate	Moderate	Growing	High	Yes	Low	Moderate
WARDuino	C	Low	Low	Arduino	Low	Yes	Low	Limited
wasm3	C/C++	Low	Moderate	Limited	Moderate	Yes	Low	Limited
AssemblyScript	TypeScript-like	Moderate	High	Growing	High	Yes	Moderate	Extensive
wasmino-core	C/C++	Low	Low	Limited	Low	Yes	Low	Limited
Binaryen	Multiple	High	Moderate	Wide	High	Yes	Moderate	Extensive
rustc (Rust)	Rust	High	High	Growing	High	Yes	Moderate	Extensive
Zigwasm	Zig	Moderate	Low	Growing	High	Yes	Moderate	Limited
fable-compiler	F#	Moderate	Moderate	Limited	Moderate	No	Moderate	Extensive
Pyodide	Python	Low	Low	Limited	Low	No	Low	Limited

Table 2. Comparison between frameworks for WebAssembly. https://doi.org/10.3390/fi15080275

Tool/Framework	Language Support	Ecosystem	Performance	Build Optimization	Learning Curve	Browser Compatibility	Development Experience	IoT Support
CheerpX	C/C++	Limited	High	High	Moderate	Moderate	C/C++ Development	Limited
Go	Go	Moderate	High	High	Low	High	Go Development	Moderate
Webpack	JavaScript	Wide	Moderate	High	High	High	JavaScript Development	Limited
Rollup	JavaScript	Moderate	Moderate	High	Moderate	High	JavaScript Development	Limited
Blazor	C#	Wide	Moderate	Moderate	Moderate	Moderate	.NET Development	Limited
wasm-bindgen	Rust	Moderate	High	Moderate	Moderate	High	Rust and JS/Wasm	Limited

2. Background

In this section, we describe the basics of Web Assembly focusing on its advantages and use cases. Moreover, we provide an overview of the Golang programming language, and its strengths compared to other languages. Finally, we present an introduction to the compiler TinyGo.

2.1. Overview of WebAssembly

> *"WebAssembly (abbreviated Wasm) is a binary instruction format for a stack-based virtual machine. Wasm is designed as a portable compilation target for programming languages, enabling deployment on the web for client and server applications."*
>
> **Web Assembly Project** [48]

2.1.1. Brief introduction

Web Assembly was first announced in 2015 and it was not until March 2017 that a first minimum viable product (MVP) was released [51]. The language quickly gained a lot of popularity and acceptance in the community to such an extent that in December 2019 World Wide Web Consortium (W3C) announced it as an official language being then the fourth language running in the browser (after HTML, CSS and JavaScript) [52].

> *"The arrival of WebAssembly expands the range of applications that can be achieved by simply using Open Web Platform technologies. In a world where machine learning and Artificial Intelligence become more and more common, it is important to enable high performance applications on the Web, without compromising the safety of the users."*
>
> **Philippe Le Hégaret, W3C Project Lead** [52]

The main purpose of Web Assembly is to enable high-performance applications to run on the Web by using a virtual machine (VM) that runs the Web application code, e.g., JavaScript code and a set of Web APIs that the Web app can call to control the Web browser/device [46].

However, since it is independent of the surrounding environment, it can be executed outside of browsers in different environments. This can be achieved with the Web Assembly Interface (WASI) and the Web Assembly singleton runtimes which we will present in the following sections.

Figure 1. Web Assembly logo.
https://www.w3.org/TR/wasm-core-1/

2.1.2. WebAssembly vs JavaScript

JavaScript was created in 1995 and was not created to be fast. However, with the increased competitiveness of browsers in 2008 many browsers incorporated JITs (just-in-time compilers) which increased the speed of JS x10. With this improvement in performance, they started to use JavaScript for cases that no one expected such as server-side programming with Node.js [7].

In Table 3 we can see how WebAssembly, in many cases, will outperform JavaScript in performing the same task. However, WebAssembly is not meant to replace JS but rather JavaScript has certain fundamental flaws that WASM was designed to address. In fact, in table x we can see how WASM takes advantage of some JS functionalities such as the Garbage collector or the access to APIS from JS.

So, WebAssembly will not replace JavaScript in the near future [20].

	JavaScript	WebAssembly
Loading Time	Javascript can be incredibly fast for smaller tasks.	WASM is perfect for heavy computation applications. It also has smaller files resulting in faster loading times.
Execution Speed	Because JS is an interpreted language, it might take a while before the browser fully understands what it's about to execute.	WASM's strongly typed code is already optimized before getting to the browser, making execution quite faster.
Garbage Collection	JavaScript has an extensive garbage collection process used for memory management.	WASM relies on JS for garbage collection.
Memory Usage	Again, JS garbage collection automatically takes care of memory usage.	Memory usage is quite complex in WASM. Developers get linear memory allocations which they have to manage manually.
Platform API Access	Javascript has extensive platform API access.	WASM doesn't have direct API access. All DOM manipulation has to be done via JS.
Debugging	As an interpreted language, debugging happens during runtime, which may appear faster but may allow errors and vulnerabilities to slip through the cracks.	As a compiled language, debugging occurs before compilation, meaning the code doesn't compile if errors are found.
Multithreading	As a single-threaded language, Javascript relies on web workers for multiple executions.	As of now, there's no Multithreading support for WASM. You can, however, use other low-level languages with Multithreading capabilities.
Portability	Developers can use JavaScript for multiple use cases and across various platforms.	WASM was built with portability in mind. Developers can use it in and outside browser platforms.

Table 3. Comparison between JavaScript and WebAssembly. https://snipcart.com/blog/webassembly-vs-javascript

2.1.3. General Features

Before moving on to the WASM specifications and some more complex definitions, we list below a brief introduction of its general features [51][38]:

- It is a binary statically compiled code base.
- Fast, small, efficient, and portable because its bytecode.
- Enable high-performance applications on web pages.
- Aims to support any language on any operating system, hardware, or browser.
- Can be run on browser with JavaScript.
- Low level enables other languages to be compiled to it: Rust, C, Golang, etc.
- It's secure with a sandbox environment.
- It has a debugging interface with human-readable text format.

2.1.4. Web Assembly Goals

The main objectives of the WebAssembly project are as follows [46][37]:

- **Fast, efficient, and portable**: WebAssembly code can achieve execution speeds close to that of native code on various platforms by taking advantage of common hardware capabilities.
- **Readable and debuggable**: despite being a low-level assembly language, has a .wat text format that allows .wasm code to be reviewed, written and debugged.
- **Secure**: it's designed to operate in a secure and isolated environment. Like other web code, it adheres to browser permissions and same-origin policies.
- **Don't break the web**: WebAssembly has been designed for seamless integration with other web technologies and to preserve backward compatibility.
- **Take advantage of underlying hardware**: WASM uses an Abstract Syntax Tree (AST) in binary format, which allows it to leverage hardware and can run on a variety of devices, IoT, Web and mobile.
- **Support non-browser embeddings**: finally due to its speed and efficiency many developers have started to use it outside of the Web, so the project is starting to support and release standards for non-browser environments.

Figure 2. WebAssembly Goals.
https://geekflare.com/webassembly-wasm-goals-key-concepts-use-cases

2.1.5. Web Assembly Specification

A WebAssembly (Wasm) program is designed as a self-contained module that encapsulates a series of Wasm-defined values and program type definitions [51]. A module contains functions, globals, a linear memory, and an indirect call table. Also, it declares all the imports and exports.

WebAssembly is defined and runs on a stack-based virtual machine. So, every type of instruction pushes and/or pops a certain number of i32/i64/f32/f64 values to/from a stack. This VM manages the evaluation stack, globals and locals [46].

2.1.5.1. Types

In WebAssembly global variables, local variables, arguments, and function results are typed with only four primitive types. These are:

- 32-bit integers (*i32*).
- 64-bit integers (*i64*).
- 32-bit floats (*f32*).
- 64-bit floats (*f64*).

More complex types, such as arrays, registers or designated pointers will be reduced to these primitive types during compilation [31].

2.1.5.2. Control Flow

WebAssembly implements a structured control flow, which means that instructions within a function are organized in well-nested blocks. Therefore, it limits the possibilities of uncontrolled jumps such as unconditional jumps, "*gotos*" to arbitrary addresses and the execution of data in memory as bytecode instructions. This contributes to higher security by preventing classical attacks such as injecting shellcode or abusing unrestricted indirect jumps [31].

2.1.5.3. Indirect Calls

WebAssembly employs indirect calls to implement function pointers and virtual functions. In Figure 3 we can see this behavior by means of the "*call_indirect*" instruction. It extracts a value from the indirect call table which corresponds to the index of a function and then calls it. In this way a function can be called indirectly if it is present in the table.

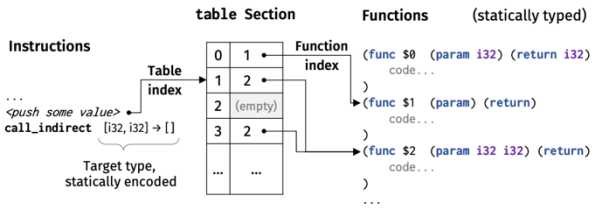

Figure 3. Indirect function calls via the table section. https://www.usenix.org/conference/usenixsecurity20/presentation/lehmann

2.1.5.4. Linear Memory

As for memory, WebAssembly provides only a linear array of bytes read and written by the low-level memory access instructions and addressed by 32-bit pointers. You can ask the VM to increase this memory with the "*memory.grow*" instruction and the organization of the sections of this memory (such as stack, heap, data) is given by the compiler itself [31]. We will comment more details of this memory in the next sections [31].

2.1.5.5. Host environment

WebAssembly modules run in a host environment that provides them with different functionalities, such as I/O or network access. These functions must be imported into the WASM module. Thus, in browsers, it is possible to import all APIs accessible to JavaScript-based client web applications, such as "*document.write*". Outside the browser, we have Node.js for server-side applications or we can use a standalone virtual machine that through the WASI standard will access to different operating system functions such as the file system [31].

2.1.5.6. Compilers

Although .wasm code can be written with the .wat text format, it is not intended to be written, but to be executed and compiled from another language. Nowadays there are several source languages such as Go, C, Rust, C++, Python, and so on. Not all compilers support all these languages, for example, Golang can only be compiled through TinyGo. In addition to the language, each compiler adds its implementations and standard libraries [31].

2.1.5.7. Code representation

In Figure 4, you can see an example of a C function that has been compiled to Web Assembly. Although the .wasm code is binary (as you can see on the right of the figure) it can be converted to a human readable .wat text format (as you can see in the center of the figure). This format is quite useful to analyze the result of the compilation.

C source code	WebAssembly `.wat` text format	WebAssembly `.wasm` binary format

`int factorial(int n) {`
`  if (n == 0)`
`  return 1;`
`  else`
`  return n * factorial(n-1);`
`}`
	`(func (param i64) (result i64)`
	`  local.get 0`
	`  i64.eqz`
	`  if (result i64)`
	`  i64.const 1`
	`  else`
	`  local.get 0`
	`  local.get 0`
	`  i64.const 1`
	`  i64.sub`
	`  call 0`
	`  i64.mul`
	`  end)`
		`00 61 73 6D 01 00 00 00`
		`01 06 01 60 01 7E 01 7E`
		`03 02 01 00`
		`0A 17 01`
		`15 00`
		`20 00`
		`50`
		`04 7E`
		`42 01`
		`05`
		`20 00`
		`20 00`
		`42 01`
		`7D`
		`10 00`
		`7E`
		`0B`
		`0B`

Figure 4. Representation of the code in C, .wat and .wasm formats.
https://en.wikipedia.org/wiki/WebAssembly

2.1.6. Web Assembly Interface

Web Assembly Interface (abbreviated WASI) is a modular collection of announced standardized APIs first announced in March 2019 [9]. This API was designed by the Wasmtime project to make WASM independent of browsers, Web APIs, and JavaScript by allowing access to different operating system features such as the file system, clocks, random numbers, and more [4].

Because the core WebAssembly language is independent of the surrounding environment, i.e., it is an assembly language for a conceptual, not physical machine, it can run on a wide variety of different machine architectures [9]. The problem is that WASM communicates exclusively through APIs [4] and until now there was no standard that facilitated the creation of truly portable WASM programs outside the Web.

WASI emerged as a solution to this need and, since WASM is a conceptual machine, it needs a system interface to a conceptual operating system. Thus, it can run on all operating systems allowing for a great deal of portability [9].

Finally, despite allowing execution of programs outside the browser, this does not disable execution in browsers because it has a polyfill that runs in browsers. In the end WASI is a set of callable functions that are imported into a .wasm module, so these functions can also be imported and utilized within JavaScript code through the use of a polyfill [4].

Figure 5. WASI icon. https://blog.vasquezruiz.me/wasi-una-interfaz-de-sistemas-para-webassembly/

2.1.6.1. WASI framework development

The standard is currently under development and consists of the following phases [47]:

- **Preview1**: Support existing users, portability.
- **Preview2**: Rebase WASI on Wit.
- **Preview3**: Level up Async: future and stream.
- **WASI 1.0**: Standardization.

At the moment, we have "*Preview1*" phase available, but the WebAssembly Community Group is working on "*Preview2*". This version will add more functionalities like sockets, time zones or file locking, which means that currently these features are not available.

2.1.7. Web Assembly Runtimes

As mentioned above, WASM is a binary language, and its execution behaviour is defined through an abstract machine that represents the state of the program. This abstract machine models how WebAssembly instructions manipulate data and control flow within a program.

WebAssembly is designed to be run on a portable stack-based virtual machine (VM) [31]. This means that instead of running directly on physical hardware, WebAssembly code executes in this virtual machine, which uses a stack data structure to manage operands, control constructs and program flow.

In this context, a WebAssembly runtime is a software component that facilitates the necessary interactions between the WebAssembly stack-based virtual machine and the environment in which it operates [31]. Therefore, in order to be able to execute .wasm code outside the browser, standalone WebAssembly runtimes compliant with WebAssembly System Interface (WASI) specifications are necessary.

There are currently many active WASI-compatible runtimes projects that supports different languages. One popular project is wasmtime, a secure and fast runtime created by the Bytecode Alliance organization. On the other hand, we can also find other runtimes such as wasmedge focused on running wasm code in could native, edge or decentralized apps or WASMR (WebAssembly Micro Runtime) which has features to run in embedded systems, IoT among others.

2.1.8. Web Assembly a Promising Cloud technology

Today cloud technologies are very popular and employed by many companies. However, due to the high demands of the organizations that choose to lease these services, they have some minimum requirements. These characteristics are as follows [3]:

- Strong security.
- Small binary sizes.
- Fast loading and running.
- Support for many OS and architectures.
- Interoperability with cloud services.

The cloud started with virtual machines that despite being secure were very heavy and startup times were not fast. Then Docker appeared with containers that allowed smaller sizes with faster load times that together with Kubernetes allowed interoperability with other cloud services [3].

However, with the advent of WebAssembly, which offers security, smaller binaries, cross-OS, cross-architecture and built for interoperability makes it a promising cloud technology lighter than Docker [3].

> *"WebAssembly is coming into its own. Expect to see it in the browser, on the edge, in the cloud, and perhaps even on a light switch or doorbell! It might just be a binary format, but it's an exciting one that is opening many new opportunities."*
>
> ***Matt Butcher[1], CEO at Fermyon Technologies***

[1] https://www.linkedin.com/in/mattbutcher/

2.1.9. Web Assembly Real cases

In the following section we present two real use cases of Web Assembly inside and outside the browser.

2.1.9.1. Google Earth

One of the main objectives of the **Google Earth** team is to bring this tool to the largest possible number of browsers, since until now, Google Earth was only accessible from Google Chrome. This was because the tool used Native Client (NaCI), a Chrome-only solution that ensured the smooth execution of the program in the browser [28].

However, thanks to the recent WebAssembly technology, in February 2020 the head of Google Earth announced that the software was now accessible from Firefox, Edge and Opera browsers [28].

Thanks to WebAssembly, a giant step towards this goal has been taken, as it would otherwise involve creating new code for each browser, which would be time-consuming.

Currently, the Google Earth team is still working on bringing Google Earth to more browsers such as Safari [28], with the help of Web Assembly.

Figure 6. Google Earth's logo. https://es.wikipedia.org/wiki/Archivo:Logotipo_de_Google_Earth.png

2.1.9.2. Shopify

Shopify is an ecommerce platform with which companies and individuals can create their online store to sell their products. It is a cloud solution (SaaS) and is one of the most popular and widely used ecommerce platforms worldwide [35].

In December 2020, the company announced that it is using WebAssembly outside the browser to bring more security, performance and flexibility to e-commerce. Shopify uses Lucet as runtime and wasm compiler. When a web request arrives (for example creating a discount) Shopify calls the Wasm Engine and then it applies the output in context of the callsite [27] as you can see in Figure 11.

With this example we can see how large companies are starting to incorporate WASM in their solutions taking advantage of the benefits offered by this language.

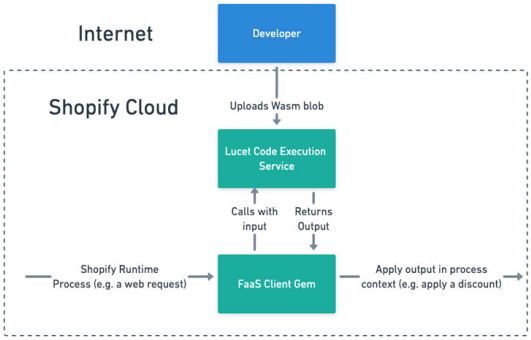

Figure 7. Shopify Wasm Engine. https://shopify.engineering/shopify-webassembly

2.2. Overview of Golang

Golang is an Open Source programming language developed by Google in 2009 [10]. It is a language inspired by C, but with memory safety and garbage collection. Some outstanding features of Golang are as follows [10]:

- It's an **Open Source** project.
- Its syntaxis is like C.
- Go is **static type checking**, which means that the type checking is performed during compilation, not during execution time.
- Go is a **compiled** language like C and C++.
- Go provides a **garbage collector**, **reflection** and some other high-level language capabilities.
- It provides **cross-compilation**, i.e., it can create executable code for another platform than the one on which the compiler runs.
- It supports the object-oriented programming paradigm but without inheritance.
- There are no "Class" definitions but separate definitions through interfaces, structs, types and embedded values.
- Go uses concurrency through goroutines (lightweight execution thread).

Figure 8. Golang icon. https://hacks.mozilla.org/2019/03/standardizing-wasi-a-webassembly-system-interface/

2.2.1. Strengths of Go compared to C

As we have seen in the features Golang is similar to C, but at the same time it is quite different. The Golang project aims to be "*as easy to use as a scripting language, but as fast as a compiled language*" [25]. It achieves this mainly thanks to automatic memory management (through the garbage collector), type safety, an integrated set of developer tools and concurrency management through goroutines. These features make programming in GoLang much simpler than with C.

For example, in the case of C we have to worry about [25]:

- Freeing the allocated memory while in Golang the garbage collector takes care of it automatically.
- Initialize the variables to zero so that they do not acquire the "garbage" value in memory. In Go they are automatically initialized to zero.
- We can't return the address of a local variable outside the function, since it would be in the "free" section of the stack and its value could be lost. In GoLang this local variable would be passed to the heap, with the automatic functionality escape to the heap [21].
- On the other hand, concurrency management is much simpler with golang, in C we would have to worry about using synchronization primitives, such as semaphores, locks, and condition variables. However, Go uses a memory model where "*Do not communicate by sharing memory; instead, share memory by communicating*." [25].

Even the syntax in Golang is much simpler, since [25]:

- It is not necessary to use parentheses in flow structures.
- No need for the final semicolon ";" at the end of a semicolon.
- No pointer arithmetic which makes it more secure by not allowing access to illegal addresses.
- It uses a backwards declaration where the expression and type syntax are separated.

However, despite all these advantages there are also some disadvantages such as [11]: the Go size is bulkier than C or the Go performance is slower than C due to the GC, in C you decide when to free the memory while in Go the compiler decides when to execute a GC cycle which reduces the speed.

Personally, having used both languages, I believe that the advantages of Go outweigh the disadvantages, mainly because its libraries, syntax, memory management and concurrency-based model make development much easier, and it is still a fairly fast language compared to other compiled languages.

2.2.2. Real-world applications implemented with Go

Many software applications have chosen to use Golang as their deployment language, such as Prometheus, Grafana, Terraform, among others. However, among the applications written in Golang stand out Docker and Kubernetes used to create and orchestrate environment-independent containers to facilitate portability, run applications in the cloud, scale, among other functions [49].

In the case of Docker, Docker has employed GO because of "*Go, with its ability to execute low-level system calls, made building Docker faster and less time-consuming than other programming languages*"[49]. In addition to that, Docker's efficient virtualization is enhanced by the use of the Go programming language.

On the other hand, Kubernetes, a container orchestrator, has been written in Golang because it is minimalistic, has good string processing, compilation, low-level system calls, and concurrency features [49].

Figure 9. Docker icon. https://www.docker.com/company/newsroom/media-resources/

Figure 10. Kubernetes icon. https://kubernetes.io/

2.2.3. Compile Golang to WASM

Having described what Golang and WebAssembly are, it is time to explain how to compile a Go code to WASM binary.

Until this year 2023, GoLang compiler had no support for WASI, only for WebAssembly through the existing *GOOS=js GOARCH=wasm* port. Without this standard it makes it difficult to run WASM code in non-browser environments using runtimes such as wasmtime.

However, on January 30, 2023, engineer Johan Brandhorst-Satzkorn has announced the implementation of WASI Preview1 with the addition of a new port *GOOS=wasip1 GOARCH=wasm*, that targets the wasi_snapshot_preview1 presented before "*Preview1*" [24].

On the other hand, we also have another project called **TinyGo** which has been implementing this target since March 5, 2021 [42]. The engineer Brandhorst-Satzkorn[2] is also part of this project which aims to bring the Golang code to different small devices.

This compiler, which also incorporates WASI before Golang's compiler, reduces the size of the binary, making it more portable and suitable especially for cloud and browser environments.

In conclusion, in this project we have chosen the TinyGo compiler since it was the first to incorporate the WASI standard and to reduce the size of the binaries. In the following sections we will see in more detail the features of this compiler.

[2] https://jbrandhorst.com/page/about/

2.3. Overview of TinyGo

TinyGo is an LLVM-based Go compiler and runtime library intended for use on small devices, Web Assembly, and command line tools [43]**.** While similar projects such as emgo[3] exists, TinyGo stand out by preserving the Go memory model like the Garbage Collector (an essential functionality). The aim behind this is that the project intends to make minimal alteration to the GoLang source code.

The initial purpose of TinyGo was to bring Go to small devices and single-core microcontrollers. As Rob Pike (a senior GoLang contributor) mentioned at the conference *Gopher Con Opening Keynote* in 2014 [12]:

> *"We never expected Go to be an embedded language and so it's got serious problems..."*
>
> **Rob Pike, Gopher Con Opening Keynote**

TinyGo was created to meet this need and currently supports up to 94 different microcontrollers. However, the project has evolved so much that it can even produce Web Assembly code that can be run on browser as well as server and edge computing environments through the web assembly interface (WASI) [40].

2.3.1. TinyGo Goals

To summarize, the main goals of this project are the following [40]:

1. Reduce the binary code size as much as possible.
2. Support for most common microcontroller boards.
3. Support WebAssembly.
4. Support CGo, with the necessary overhead.
5. Support most libraries and packages of Golang without modifications.

Figure 11. TinyGo's logo.
https://tinygo.org

[3] https://github.com/ziutek/emgo

2.3.2. Compiler Advantages and Disadvantages

Compiling Golang code with TinyGo offers many advantages, such as:

- The generated WASM file is tiny (that's why it is called Tiny Go) and its size makes it ideal for working with small hardware devices, in the browser, edge computing, IoT or cloud
- It offers different levels of optimization so that we can adjust them more or less according to our priorities, either to reduce the size or to increase the speedup.
- Compiles for different targets, and as for WASM it can compile for in-browser and out-of-browser thanks to the WASI standard.
- In general, the programming language and standard libraries are the same as in Go, except for a few listed in the compiler documentation but which are still being improved.

However, since it is a project still in the development phase, it has some drawbacks that are being worked on. These are mainly based on the incorporation of some features offered by Golang.

The main disadvantages are the following [23]:

- Not all Golang standard libraries are implemented when targeting WASI.
- Some Cgo functions are not yet supported or may work slightly differently.
- The reflect package has been implemented, but some parts are not yet fully compatible. This causes some Golang libraries to not finish compiling properly.
- The garbage collection behavior is implemented but may not work very well on WASM or small devices. In addition to being slower than Golang.

2.4. Golang to WebAssembly conversion

In Figure 12, you can see the process for performing this conversion:

1. First, we write the code in Golang.
2. Then, we compile the code with TinyGo adding WASI as a target to obtain the WASM module.
3. Finally, we will be able to run this module with a runtime on different architectures and operating systems.

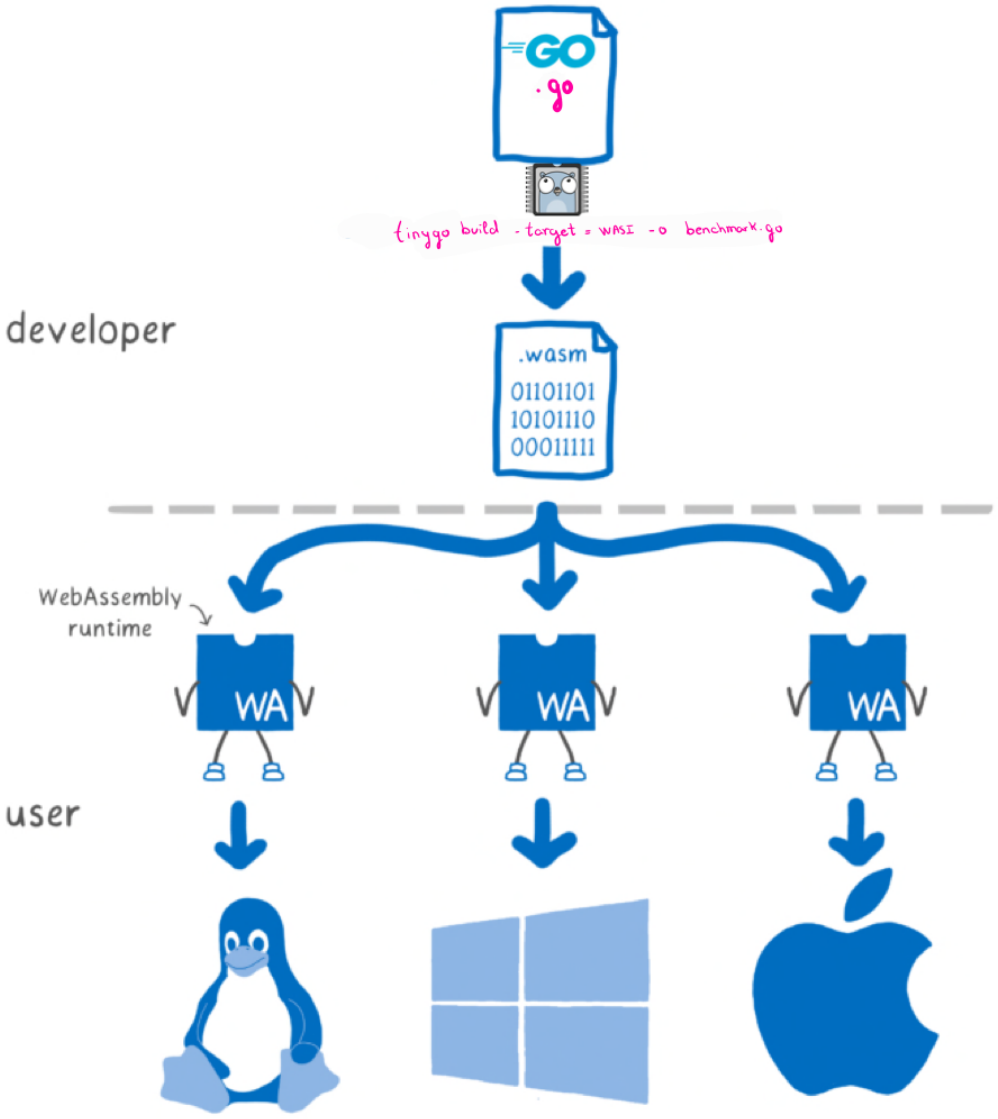

Figure 12. Conversion from Golang to WebAssembly. Figure modified from: https://www.alibabacloud.com/blog/using-webassembly-and-kubernetes-in-combination_596177

3. Memory management

Since one of our objectives is to analyze in depth the memory management system, in this section we will briefly introduce its different concepts, functions and classifications. Thus, we will focus on: i) *Basic concepts of memory management*; ii) *Memory management in Golang*; iii) *Memory management in TinyGo*; iv) *Memory management in WebAssembly*; v) *Summary of differences between memory managements*.

3.1. Basic concepts of memory management

Memory management is a way of managing computer resources. This system is mainly based on dynamically allocating portions of memory to programs when they need it and freeing it when they no longer need this space for reuse it [50].

Memory management is a very important factor, as the way space is managed can affect software performance. In general, depending on whether the allocation and deallocation of memory is done explicitly or not, we can classify the memory management system as manual or automatic [19]:

- ***Manual Memory Management***: this memory management falls on the programmer, he is the one who allocates and releases the objects in an explicit way. This management gives the programmer more control over the memory used. However, it can produce unpredictable bugs and may not be comfortable for programmers as they have to write more code.
- ***Automatic Memory Management***: this memory management is quite comfortable and safe for the programmer since it allocates and releases memory automatically without human intervention through the garbage collector. However, it may be slower, and you do not have as much control over the memory used.

3.1.1. Memory Layout

First, to understand memory management, we need to take a look at the memory layout. In Figure 13, we have a simple example composed of 4 basic elements. Normally it has more sections but to simplify the explanation we will focus on the following general elements [5].

- ***Program code***: the main program, its variables and arguments are placed here.
- ***Global***: this space is reserved for static or global variables that will be used throughout the lifetime of the program.
- ***Stack***: the stack is divided into stack-frames and will contain the information of the functions together with their local variables and arguments. The size of data in this section is known in compilation time, which means that stack is finite and static.
- ***Heap***: this sections is used for dynamic memory allocation, which means that the memory is allocated during the runtime or execution time. Heap is more flexible than stack and it normally contains pointers, arrays and big data structs. It has not specific structure or size and can be adapted.

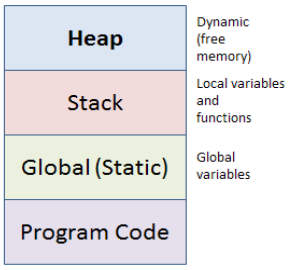

Figure 13. Basic Memory Layout. https://study.com/academy/lesson/how-to-allocate-deallocate-memory-in-c-programming.html

3.1.2. Memory Allocation

Memory Allocation is part of memory management and it's the process of reserving physical or virtual memory needed for programs or services. This process is managed by the operating system and can be classified according to the type of allocation (static or dynamic) or programing context (stack or heap).

According to the type of allocation we have [22]:

- ***Static allocation***: the data size is already known, and it's reserved in the compilation time.
- ***Dynamic allocations***: the allocation of the memory is done in time of execution or runtime.

On the other hand, according to the programming context we have:

- ***Stack Allocations***: When a function is called, a block (also known as stack-frame) is reserved. This stack-frame is important because it establishes the order of calls between routines or functions, thus allowing subroutines to be called. This is allocated at compile time and follows the "last in first out" (LIFO) approach [30].
- ***Heap Allocations***: unlike the stack, the heap memory (also known as free pool or free store) does not have a concrete way of organizing itself, so it requires us to indicate which memory will be reserved and which will be cleared. Heap memory is allocated in runtime, then its dynamic allocation. The heap is a graph where objects are represented as nodes which are referred to in code or by other objects in the heap. As a program runs, the heap will continue to grow as objects are added unless the heap is cleaned up [22].

3.1.3. Memory Deallocation

Once the program no longer needs the memory, it must be freed to reuse the space for other programs, this process is known as memory deallocation, free or release.

***Stack deallocation*:** in stack the allocation or deallocation processes are automatically done by the compiler. This process is easy and fast because the block to deallocate is already know thanks of the LIFO structure and contiguous allocations [19].

Heap deallocation: otherwise, in heap the deallocation process is more complicated since the data is allocated in any order with any size during the execution time. Also, if heap is not cleaned up, it will continue to grow as objects are added. This process of cleaning can be performed manually or automatically with some strategies such as garbage collector [19].

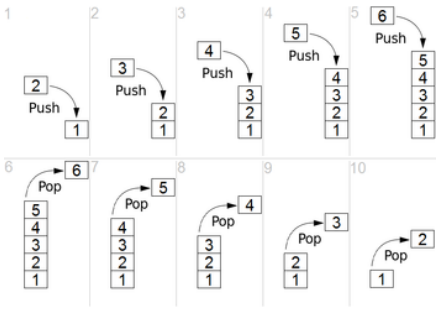

Figure 15. Stack allocation and deallocation. https://medium.com/safetycultureengineering/an-overview-of-memory-management-in-go-9a72ec7c76a8

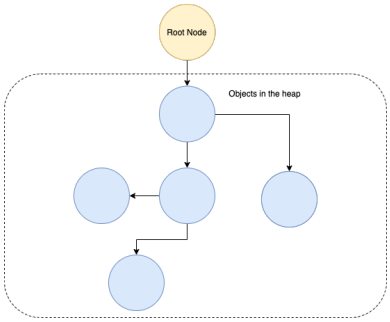

Figure 14. Heap allocation and deallocation. https://medium.com/safetycultureengineering/an-overview-of-memory-management-in-go-9a72ec7c76a8

3.2. Memory management in Golang

Go is based on an automatic dynamic memory management system through a garbage collector [19]. In this section, we present i) *Goroutines memory allocation*, ii) *Go's Escape analysis* and iii) *Garbage collector*.

3.2.1. Goroutines memory allocation

In Go, each goroutine has a stack (with a block of memory initially allocated) and a global dynamic heap accessible for all the goroutines. These are the two places where Go can allocate memory.

What makes Golang different from other garbage collector languages is that it prefers allocations in the stack because it's cheaper since only need two CPU instructions [19]. However, not all data can be placed in stack because the lifetime and space have to be determined in compilation time. Then, the heap is used instead to allocate dynamically in runtime.

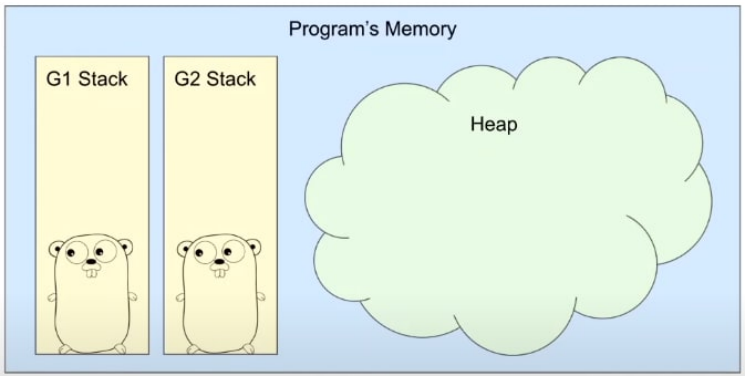

Figure 16. Go Program's Memory.
https://dev.to/karankumarshreds/memory-allocations-in-go-1bpa

3.2.1.1. Additional space required

When a goroutine requires more space that what is available in its stack (which can cause stack overflow), we can apply three possible solutions [5]:

1. Increasing the stack size, which will increase the stack size of all threads.
2. Specifying the size needed for each thread.
3. Starting with a fixed stack size and increase according to the needs.

For the first and second solutions, increasing the stack size for all threads or having to manage the space specifically for each thread is an inefficient methodology. That is because, in the first case we would be reserving more memory than necessary and in the second case we would have to find out in advance the amount of memory needed for each thread. Thus, the last solution is the best since we will not have to think about the stack size of the thread, because it will be increased according to the needs.

3.2.1.2. Increasing stack memory procedure

As we have seen, in go the size of stack is increased according to the needs. Then, goroutines start with a size of 2Kb and increases the size as needed without the risk of running out [5]. To implement this solution [6]:

1. First, the linker (5l, 6l, 8l) inserts a small preamble at the beginning of each function. This will check if the amount of stack required for the function is less than the amount currently available.
2. If the amount of memory is not enough the *runtime.morestack* is called. This method will allocate a new stack page, copies the caller's arguments, and then returns control to the original function, which can now safely execute.
3. Finally, when that function exits, the arguments are copied back to the caller's stack frame, and the unnecessary stack space is freed .

According to this procedure, it seems that the amount of memory that we can allocate in stack is infinite, but, actually, the new stack pages are allocated in heap and its memory depends on a lot of things such as the operating system or the architecture of the computer [6].

3.2.2. Golang's memory escape behavior

As we have seen before, Go prefers to allocate objects in the stack instead of the heap, but this operation is not always available, because we can't determine lifetime or spaces in compilation time, and then the object has to be stored in the global heap section.

However, there is other special cases where Go has to allocate memory in heap. This procedure is called escape behavior or escape to de heap, and it is used to deal with pointers, interfaces, or insufficient stack space [16].

3.2.2.1. Pointer scape

Let's see an example with the following Golang code [16]. We can see that we have a function called demo that returns a pointer to an integer data value. On the other hand, this function is called in the main function, and this last one prints the content of that variable on terminal.

demo.go

```
package main

func demo() (*int) {
    var data int = 11
    return &data
}

func main() {
    result := demo()
    println(*result)
}
```

If we take a look to the structure of the goroutine stack on Figure 17, we can see that main and demo functions are allocated in order in the stack (first main function is pushed to the stack and then demo). Inside the stack-frames we can see that there are the local variables for each function (data=11 and result=nil).

Once the function demo finishes or exists, returns the pointer to main of the data local variable. Then, the space in stack is automatically released to be reused. So, now, the variable "result" in the main function is pointing to space in memory which is considered "freed". Then, the real value of the "data" variable (11) can be lost and replaced with other information since it is a free space.

In order to store and save this information, Golang moves the contents of the variable to the Heap memory and returns a pointer to this memory space. Then when the main accesses the pointer location, it will find the original value of data (11).

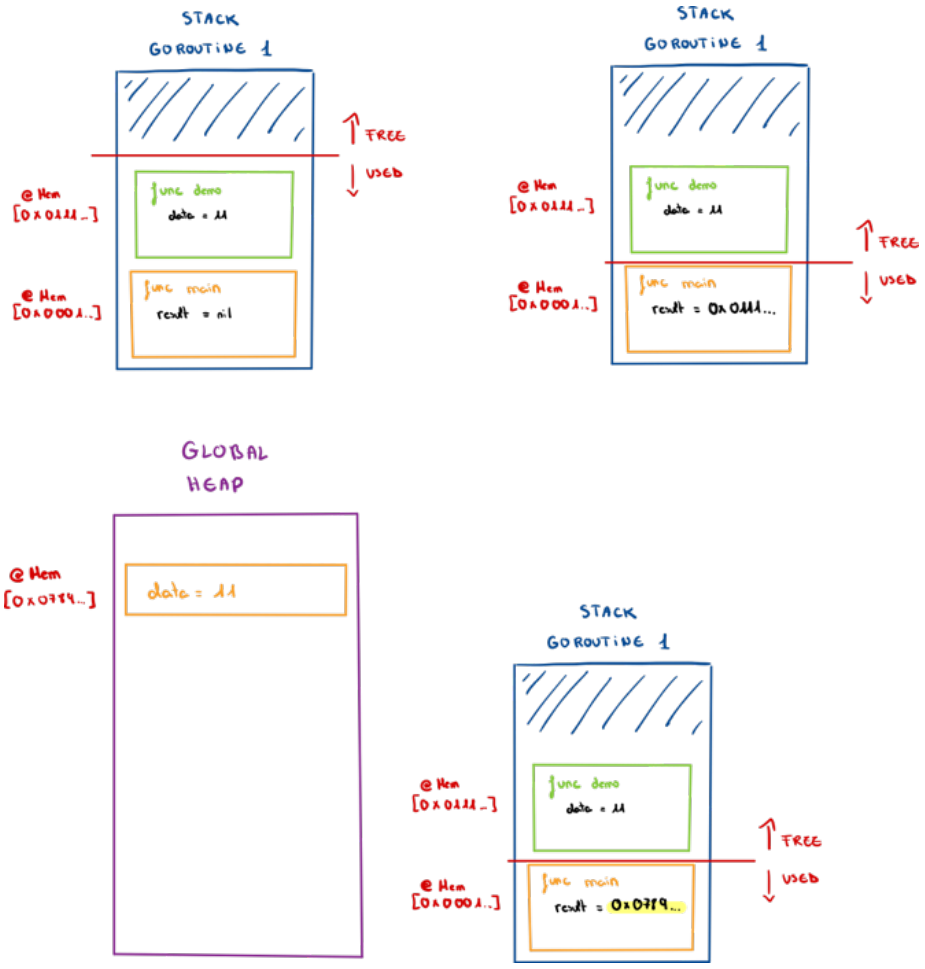

Figure 17. Escape to the heap behavior. Generated by author Safia Guellil.

Finally, if we execute the following command (*go build -gcflags '-m'*,), we can see on terminal (Figure 18) that the variable data was moved automatically to the heap to protect its value to be replaced through the escaping behavior.

```
parallels@ubuntu-linux-22-04-desktop:~/Desktop/TFM/Analysis/Escape$ go tool compile -m  demo.go
/home/parallels/Desktop/TFM/Analysis/Escape/demo.go:4:6: can inline demo
/home/parallels/Desktop/TFM/Analysis/Escape/demo.go:9:6: can inline main
/home/parallels/Desktop/TFM/Analysis/Escape/demo.go:11:19: inlining call to demo
/home/parallels/Desktop/TFM/Analysis/Escape/demo.go:5:9: moved to heap: data
```

Figure 18. Example of escape to the heap behavior .Generated by author Safia Guellil.

In fact, we can find in the official Golang documentation the following explanation that justifies when a variable is escaped to heap memory. [18]:

> ***How do I know whether a variable is allocated on the heap or the stack?***
>
> *When possible, the Go compilers will allocate variables that are local to a function in that function's stack frame. However, if the compiler cannot prove that the variable is not referenced after the function returns, then the compiler must allocate the variable on the garbage-collected heap to avoid dangling pointer errors. Also, if a local variable is very large, it might make more sense to store it on the heap rather than the stack. In the current compilers, if a variable has its address taken, that variable is a candidate for allocation on the heap.*
>
> *However, a basic escape analysis recognizes some cases when such variables will not live past the return from the function and can reside on the stack.*
>
> **Golang's website FAQ**

3.2.2.2. Insufficient stack space to escape

Finally, as we have commented in the "*Increasing memory procedure*" section when stack is full the space allocated is from the global heap instead of stack.

In the following code [16], we can see how we are trying to allocate in stack a huge slice, so Golang performs the scape analysis and moves the structure to the heap, as you can see in the Figure 19.

index.go

```
package main

func main() {
    s := make([]int, 10000, 10000)
    for index, _ := range s {
        s[index] = index
    }
}
```

```
parallels@ubuntu-linux-22-04-desktop:~/Desktop/TFM/Analysis/Escape$ go tool compile -m index.go
/home/parallels/Desktop/TFM/Analysis/Escape/index.go:3:6: can inline main
/home/parallels/Desktop/TFM/Analysis/Escape/index.go:4:14: make([]int, 10000, 10000) escapes to heap
```

Figure 19. Example of escape to the heap behavior. Generated by author Safia Guellil.

3.2.2.3. Dynamic type escape

There are other cases where the parameters of the functions are interfaces or structs that are difficult to determine the type on compilation time [16]. So, these cases also generate the escape behavior.

3.2.3. Go's Garbage Collector

Go lang has an automatic memory management, also called as garbage collector. This mechanism helps programmers to focus on the code logic instead of the memory management.

Languages that use this system have the following benefits [19]:

1. Increased security.
2. Better portability across operating systems.
3. Less code to write.
4. Runtime verification of code.
5. Bounds checking of arrays.

If we retrieve the last example (Figure 17), we can see that the pointer "*data*" has been escaped to the heap. So now, what happens if we stop needing this element? It is important to remember that if we do not clean the heap, it will continue to occupy space as objects are added. So, the function of cleaning heap objects relies on Go's Garbage Collector.

3.2.3.1. Go lang Garbage Collector

Go's garbage collector is a *non-generational concurrent, tri-color mark and sweep garbage collector* [5]. The above concepts refer to the following definitions:

- ***Non-generational***: GC is not generational since they usually focus mainly on recently allocated objects. As Go usually puts short-lived objects are often allocated on a stack, it means that the GC does not need to be generational.
- ***Concurrent***: it can run in parallel with the main program.
- ***Tri-color mark***: white represents objects that are not reached, grey the ones that are reachable and finally black the ones which are in use.
- ***Sweep***: this is the phase of freeing memory from objects that are not alive.

The trash collector consists of two parts [19]:

- ***Collector***: executes the logic and finds objects that should be released.
- ***Mutator***: executes the application code, assigns new objects to the heap, and updates them making them unreachable to be cleaned by the collector.

3.2.3.2. Garbage Collection Triggers

There are some metrics that triggers garbage collection process when [36]:

4. The heap doubles its size.
5. The GC has not been triggered for more than two minutes.
6. The *runtime.mallocgc* function can triggers GC because will create objects that can trigger a new garbage collector cycle.
7. It's manually triggered by using *gc*() function.

3.2.3.3. Implementation of Garbage Collector in Go

Go's Garbage Collector has the following 4 steps [36]:

1- ***Mark Setup***: In this first step the everything is stopped using a process called **stop the word**. Then, the **write barrier** is turned on in all goroutines to stop writing on the heap maintaining data integrity. This will stop the application.

2- ***Marking***: then, the collector will inspect global, heap pointers or the stack of each goroutine to find the root pointer to heap and traverse it to see if it is still in use or reachable. In this case reachable means that other reachable memory contains a pointer to this reachable object, i.e., a global variable contains a pointer to it, a local variable in one of the goroutines or another reachable object contains a pointer to it [1]. This marking phase is recursive [1] where Go uses around the 25% of the available goroutines to perform this operation.
The inspections start from the root which is the starting point where other heap objects come from. All found objects will be marked as grey by traversing downwards from the root. So then, these gray objects are enqueued to be turned black which indicates that are in use.

3- ***Mark Termination***: Once the marking process finishes, the collector calls again the stop the world function, puts the write barrier off and performs some cleaning. Then, calculates the schedule for the next garbage collection. So, when STW and write barriers are off, the application can start working again.

4- ***Sweeping***: Finally, the process of sweeping claims that non-marked memory allocations. This process does not add latency to the collection process because it's executed when a new allocation happens.

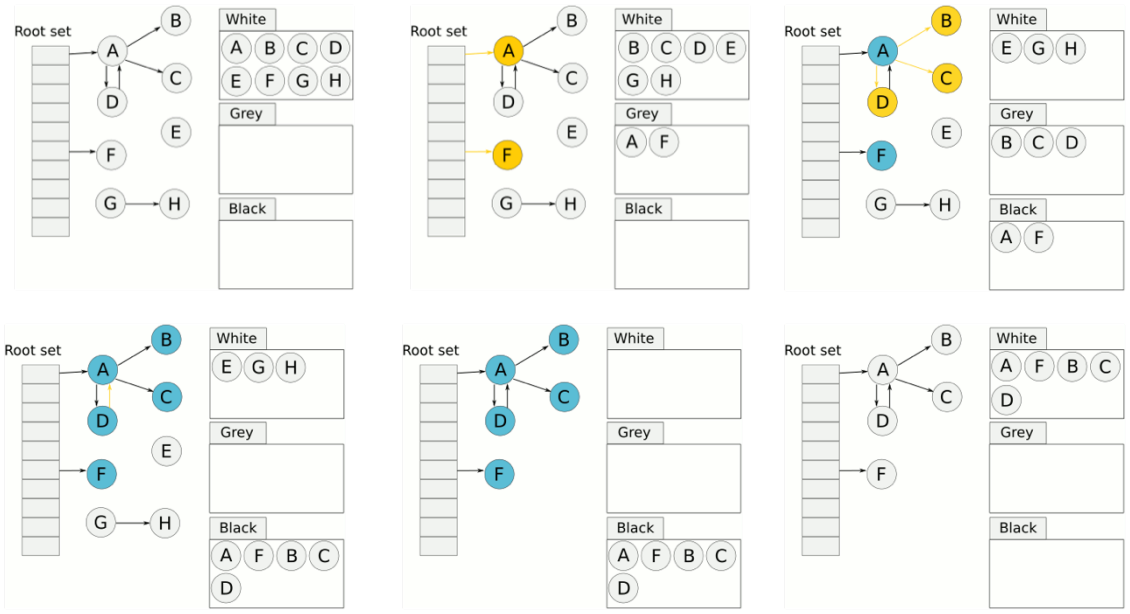

Figure 20. Go's Garbage Collector process. https://en.wikipedia.org/wiki/Tracing_garbage_collection

3.2.3.4. Example of Go's Garbage Collector

In the following code [14], we are allocating an int slice with a length of 900,000. So, due to its size, it will be allocated on the heap.

This code is printing the memory's stats before and after allocating a big object. As we have seen in the "*Garbage Collection Triggers"* section, if heap doubles its size, it will execute the garbage collector algorithm. So, this behavior is demonstrated in the output (see Figure 21) where a lot of objects have been freed and one garbage collector's cycle has been performed.

collector.go

```
package main
func main() {
        var ms runtime.MemStats
        printMemStat(ms)
        intArr := make([]int, 900000)
        for i := 0; i < len(intArr); i++ {
                intArr[i] = rand.Int()
        }
        time.Sleep(5 * time.Second)
        printMemStat(ms)
}

func printMemStat(ms runtime.MemStats) {
        runtime.ReadMemStats(&ms)
        fmt.Println("---------------------------------------")
        fmt.Println("Memory Statistics Reporting time: ", time.Now())
        fmt.Println("---------------------------------------")
        fmt.Println("Bytes of allocated heap objects: ", ms.Alloc)
        fmt.Println("Total bytes of Heap object: ", ms.TotalAlloc)
        fmt.Println("Bytes of memory obtained from OS: ", ms.Sys)
        fmt.Println("Count of heap objects: ", ms.Mallocs)
        fmt.Println("Count of heap objects freed: ", ms.Frees)
        fmt.Println("Count of live heap objects", ms.Mallocs-ms.Frees)
        fmt.Println("Number of completed GC cycles: ", ms.NumGC)
        fmt.Println("---------------------------------------")

}
```

```
parallels@ubuntu-linux-22-04-desktop:~/Desktop/TFM/Analysis/Escape$ go run collector.go
---------------------------------------
Memory Statistics Reporting time:  2023-07-06 21:51:14.636157238 +0200 CEST m=+0.000184496
---------------------------------------
Bytes of allocated heap objects:  47288
Total bytes of Heap object:  47288
Bytes of memory obtained from OS:  7191568
Count of heap objects:  150
Count of heap objects freed:  2
Count of live heap objects 148
Number of completed GC cycles:  0
---------------------------------------
---------------------------------------
Memory Statistics Reporting time:  2023-07-06 21:51:19.651556721 +0200 CEST m=+5.015584062
---------------------------------------
Bytes of allocated heap objects:  7264392
Total bytes of Heap object:  7269384
Bytes of memory obtained from OS:  15907856
Count of heap objects:  204
Count of heap objects freed:  32
Count of live heap objects 172
Number of completed GC cycles:  1
---------------------------------------
```

Figure 21. Garbage Collection Triggers. Generated by author Safia Guellil.

3.3. Memory management in TinyGo

Having analyzed the memory management system of Go it is time to analyze how the TinyGo compiler does it mainly with the garbage collector and escape to the heap behavior. These features are very important to analyze because they will help us understand which Golang functionalities are maintained in TinyGo and how they are reduced or treated. This way we will know how they will later be used in WebAssembly.

3.3.1. Memory Layout

Below, in Figure 22, you can see the RAM memory layout that TinyGo usually uses in a microcontroller. This is organized as follows (from bottom to top) [1]:

- ***Stack***: the call stack is placed at the bottom of the memory to provide hardware checking for stack overflows (even on hardware without special support for it).
- ***Data***: in the .data section we find global variables that are not set to zero.
- ***BSS***: on the other hand, in the .bss section we find global variables initialized to zero.
- ***Heap***: finally, the entire upper space is the heap section.

Figure 22. TinyGo Memory Layout. https://aykevl.nl/2020/09/gc-tinygo

3.3.2. Memory management strategies

TinyGo offers different strategies to manage memory. These strategies will depend on the programmer's needs and objectives. [1]:

- When you need to allocate memory during initialization you can use a **simple bump-pointer heap** implementation.
- When you want to work with an external heap (like malloc in C) you can use a **heap** implementation.
- But the most used in microcontrollers (BareMetal systems is a conservative **mark/sweep garbage collector**).

This last strategy, garbage collector, is the one that interests us in this project since it is the most popular and the default one that GoLang uses.

3.3.3. TinyGo's Garbage Collector

The GC of TinyGo is much simple than the Go's one, that is because it is thought to be run on microcontrollers devices. This garbage collector is *conservative* and *mark/sweep*, which means [1]:

- ***Conservative***: the compiler doesn't know exactly what it's a pointer and what not. Knowing which object is a pointer it's important for finding the root pointer and then detecting whether or not an object is reachable.
- ***Mark/Sweep***: as Go's GC, TinyGo also has a marking phase and a sweeping one where the objects are marked and freed.

As we can see, the general idea of the GC is to first mark the reachable objects, then it will follow these pointers recursively marking more and more reachable objects, and finally it will eliminate the rest of the objects that have not been marked.

Reachable objects are active objects, that are still in use. For instance, we consider reachable objects if [1]:

- They are pointed by a global variable.
- They are pointed by a local variable (stack).
- There are another reachable object pointing to them.

3.3.3.1. TinyGo's GC types

So, if we take a look to the TinyGo Git repository[4], we can see that they have published different Garbage collector implementations, that can be selected from the terminal at the time of compiling. These are the following [41]:

- ***gc_none***: does not implement the garbage collector or allocate memory on the heap.
- ***gc_leaking***: it allocates memory on the heap but does not free it.
- ***gc_conservative***: is the default implementation of the block-based heap as a fully conservative GC. It does not recognize exactly what is or is not a pointer, so it considers an object to be active if it looks like a pointer.
- ***gc_precise***: implements the block-based GC as a partially precise GC, i.e., in most cases it knows what is or is not a pointer. This new implementation has fewer false positives than the "conservative" version and makes the GC somewhat faster. However, it uses slightly more RAM.
- **gc_*custom***: finally, this implementation allows the plug in of a custom garbage collector.

[4] https://github.com/tinygo-org

3.3.3.2. TinyGo's Garbage Collector main code

Below, you can see the main code of a GC cycle (for ease of analysis we have removed some comments and some debugging code). We have highlighted the main functions that are called in a GC cycle. As we can see, this implementation has the same phases as the one used by Golang:

1. ***markStack and markGlobals***: these functions mark all root pointers found on the stack and globals which are always reachable.
2. ***markRoot***: reads all pointers from beginning to end and if they look like a heap pointer and are not marked, it marks them and scans that object.
3. ***finishMark***: it finishes the marking process by processing all ***stack overflows*** which is set when the GC scans too deep while marking so all marked allocations must be re-scanned trough startMark function. For example, if the stack is full, it is necessary to rescan all the marked blocks once we have finished.
    - ***startMark***: starts the marking process on a root and all its children. It also scans each object (this function is inside the *finishMark* so we can't see it in the attached code).
4. ***sweep***: finally, it goes through all memory and frees unmarked memory.

```
func runGC() (freeBytes uintptr) {

        markStack()
        markGlobals()

        if baremetal && hasScheduler {
                // Channel operations in interrupts may move task pointers around while we
                // are marking. Therefore, we need to scan the runqueue seperately.
                var markedTaskQueue task.Queue
        runqueueScan:
                for !runqueue.Empty() {
                        // Pop the next task off of the runqueue.
                        t := runqueue.Pop()
                        // Mark the task if it has not already been marked.
                        markRoot(uintptr(unsafe.Pointer(&runqueue)),
                        uintptr(unsafe.Pointer(t)))

                        // Push the task onto our temporary queue.
                        markedTaskQueue.Push(t)
                }
                finishMark()
                // Restore the runqueue.
                i := interrupt.Disable()
                if !runqueue.Empty() {
                        // Something new came in while finishing the mark.
                        interrupt.Restore(i)
                        goto runqueueScan
                }
                runqueue = markedTaskQueue
                interrupt.Restore(i)
        } else {
                finishMark()
        }
        // Sweep phase: free all non-marked objects and unmark marked objects for
        // the next collection cycle.
        freeBytes = sweep()
        return
}
```

3.3.3.3. Advantages and Disadvantages

Tinygo's garbage collector has some advantages and disadvantages, compared to other more complex GCs such as Golang, Java, Chrome, and so on. However, we have to consider that Tinygo's GC is thought to be run on microcontrollers and embedded systems [1].

- It's very small, the code is short.
- It's slow, with sometimes visible pause times, instead Go's GC does less pauses which has better performance.
- It's conservative which means that non-pointer data, such as integers slices, can be scanned for pointers, allowing false positives.
- As Go's GC, the memory never moves, so if the heap is fragmented, we can't compact it to have larger free area.
- It can't deal with virtual memory.

3.3.4. TinyGo's Heap Allocation

In general, GoLang performs many heap allocations, so it is an important functionality to consider in TinyGo. However, although it is implemented, it is not always possible. For example, the following operations escape the heap [26]:

- Starting goroutines.
- Reading and modifying maps.
- Creating an interface with a value larger than a pointer
- Concatenating strings, (unless one of them is zero length).
- Converting a byte or rune into a string.
- Converting between string and []byte, However, there is an optimization that avoids a heap allocation.

On the other hand, we have other cases in which the code takes the pointer of a local variable that can be escaped or not depending on the resulting pointer. For example, in the case of Figure 24 we see how the code escapes when trying to access a pointer of a local variable. While in Figure 24, although the local function bar receives a pointer, when trying to print the value of this pointer does not escape this variable to the heap.

```
var global *int

func foo() {
    i := 3
    global = &i
}
```

Figure 24. Escape to the heap in TinyGo. https://tinygo.org/docs/concepts/compiler-internals/heap-allocation/

```
func foo() {
    i := 3
    bar(&i)
}

func bar(i *int) {
    println(*i)
}
```

Figure 24. Example where the pointer doesn't escape to the heap on TinyGo. https://tinygo.org/docs/concepts/compiler-internals/heap-allocation/

3.4. Memory management in Web Assembly

WebAssembly does not have any memory management system [17], it only provides a linear memory which is a simple global array. However, this linear memory needs a system to manage it. In the following section we will analyze this memory, how it is organized and future proposals to add a management system in WebAssembly.

3.4.1. Managed and Unmanaged Data

In the context of WebAssembly, we can distinct between managed and unmanaged data [31]:

- ***Managed data***: these are local variables, global variables, evaluation stack values and return addresses, is hosted in specific storage managed directly by the virtual machine. WebAssembly code can only interact with this data through instructions, but not directly modify its underlying storage which is not visible.
- ***Unmanaged data***: on the other hand, this data is what is in the linear memory of Web Assembly. All non-scalar data such as strings, arrays or lists are stored here, and they are completely under the control of the program and is organized by code generated by the compiler.

Managed data has no address, therefore any variable whose address is taken in the source program will be stored in linear memory. Since numerous non-scalar types are present in the source code as function, global or dynamic data, the compiler generates segments within the linear memory to accommodate a call stack, a heap, and static data, organizing the linear memory like the typical memory layouts we have seen in theory [31].

In the following sections we will focus on the unmanaged data and the linear memory of WASM since these are the ones that live given by the compiler.

3.4.2. Linear Memory Layout

As we have seen in the initial background section, the WebAssembly linear memory is simply a global array of bytes. However, this memory can be structured and organized with typical sections (.data, .bss, heap, etc.) depending on the compiler. For example, in the following Figure 25 you can see how different compilers organize the WebAssembly linear memory.

In this case, we can observe how the stack or data positioning and the growth of these towards high or low directions differs between the compilers. However, the heap is always located in the upper part of the line memory, this is because it can grow in case more memory is required [31].

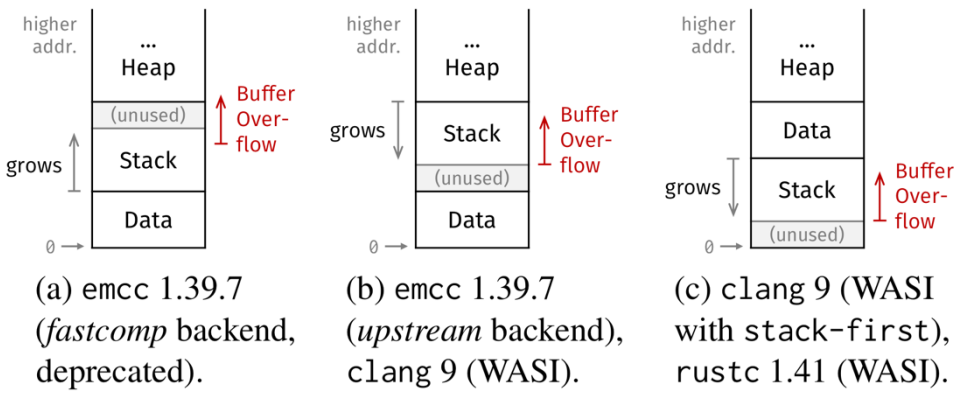

Figure 25. WASM linear memory for different compilers. https://www.usenix.org/system/files/sec20-lehmann.pdf

3.4.3. WASM's Garbage Collector Proposal

Since 2017, the WebAssembly team has intended to add a garbage collector that gives WebAssembly access to the built-in GC with a set of low-level GC primitive types and operations. This is basically because WASM did not know how to interact with the existing garbage collector, mainly the one provided by the JavaScript engine [8].

Until now languages that need GC have had no choice but to compile their own GC code and add it to the binary. This causes an additional cost in both performance and binary size [17]. There is currently a proposal to add a garbage collector to WebAssembly. It is a complex proposal that is adding several significant changes to WebAssembly. For example, the addition of type systems including tuples, structures and simple arrays and even more proposals such as reference types are emerging. The latter is an important component in the construction of the garbage collector.

However, this proposal will not cause WebAssembly to have its own garbage collector but will be integrated with the GC provided by the host environment.

As of today, the proposal is not yet finished, since as we have mentioned it is a complex proposal that requires the addition of other sub-proposals. However, the repository is still very active[5].

[5] https://github.com/WebAssembly/gc

3.5. Conclusion about Memory Management

Finally, after analyzing the memory management systems of this 3 elements: GoLang, Tinygo and WASM (although the latter has none), we can generate some conclusions and establish some research objectives.

We will focus on the three fundamental aspects or differences that we have found during this research phase; these are: the memory layout, the garbage collector, and the escape to the heap behavior. Thanks to this analysis, we can establish some first premises to analyze in the following section Evaluation.

3.5.1. Memory Layout

As we have seen in the theory, the linear memory structure of WebAssembly is given by the compiler. It is generally structured by the heap, stack and data sections, whose organization and growth (to higher or lower memory addresses) are specified by the compiler.

Although TinyGo briefly specifies how memory is organized, we would like to verify that it is indeed organized in such a way in WebAssembly. We would also like to check in which direction the heap and stack grow within the linear memory.

3.5.2. Escape to the heap

Finally, escape to the heap is a prominent feature in Golang that prevents us from future errors by automatically moving data to the heap. As we have seen, this functionality is maintained in the TinyGo compiler, but unlike Golang, it does not always end up working in practice.

On the other hand, there is no reference to this functionality in WebAssembly, which would be interesting to analyze and check if this feature is maintained, i.e., that the elements that have escaped to the heap are in the heap section of the WASM linear memory.

3.5.3. Garbage collector

Finally, as for the Garbage Collector, we have seen that it is a central feature in the Golang language since its memory management model is mainly based on it.

On the other hand, TinyGo also implements a GC, but it is much simpler and conservative, since it does not detect well what is or is not a pointer, leading to false positives. Therefore, unlike Golang the GC is expected to be slower and even with visible pauses.

However, TinyGo continues to improve, and they have released a new conservative, but more accurate version that could increase the speed of the GC but would use more RAM.

Finally, as for WebAssembly, it does not have a garbage collector although there is an active proposal that would add GC support. In fact, so far, the languages that have needed it have had to compile it, thus increasing the size of the binary.

Therefore, in the case of GoLang, it is expected that the TinyGo compiler will also compile the GC code to binary. In this case, the GC we will see will be the simpler version offered by TinyGo. In addition to that, it would be interesting to compare both GCs and observe the differences in both time and size of the different GC options offered by TinyGo.

4. WebAssembly Evaluation

In this section, we evaluate the compilation of Golang code with TinyGo to generate a WASM binary. This evaluation has been done in terms of memory management (memory layout, escape to the heap, and garbage collector) and performance (execution times, binary size, and system calls). So, the evaluation is divided as follows: i) *Evaluation questions and methodology; ii) Evaluation of memory management; iii) Evaluation of Performance.*

4.1. Evaluation questions and methodology

First, before we begin the evaluation, we need to establish some evaluation questions. These questions will help us focus our efforts on analyzing specific desired features of the compiled code.

4.1.1. Methodology for assessing memory management

As we discussed in the background, one of the outstanding features of Golang is the dynamic and automatic memory management system. In the following sections, we analyze how this memory management would look like after compiling it to Web Assembly using TinyGo. This analysis focuses on these three elements detected during the memory management overview done in the last section:

Q1. Analyzing the Linear Memory Layout Structure

As we have seen earlier in the memory management section, the structure and organization of Web Assembly's linear memory depend on the compiler. In this case, we have seen how the TinyGo compiler is organized, but we have not found an example that demonstrates this organization in Web Assembly's linear memory. Furthermore, we lack information on how stack addresses grow. Therefore, our first research tries to:

1. Demonstrate the implementation of this organization in WASM.
2. Identify the address growth pattern of the stack.
3. Finally, compare the organization of linear memory with other compilers.

Q2. Analyzing the Escape to the heap behavior

Another important and remarkable feature of the Golang language is the escape to the heap behavior. In this section, we analyze two different cases where variables have escaped from the stack to the heap. These cases are:

1. Accessing the memory address of a variable.
2. Allocating an object larger than the available stack space.

These cases have been previously shown in the Golang Memory Management section, and we have seen how they have indeed escaped to the heap. In this section, we will focus on analyzing whether they also escape with TinyGo and WebAssembly.

Q3. Analyzing the Garbage Collector

Finally, we analyze the garbage collector mechanism. As mentioned in the background, since WebAssembly lacks built-in garbage collection, languages that have needed it have had to compile it and add it to the .wasm module. In this section, we examine:

1. Determine if TinyGo's garbage collector is added to the Web Assembly binary.
2. Compare the number of automatic garbage collector cycles between Golang and TinyGo, considering the different TinyGo's GC options.
3. Manually trigger the garbage collector and measure the time spent on one cycle.

4.1.2. Methodology for assessing performance

After evaluating the memory, we need to evaluate the performance of the module compiled with WebAssembly. This performance evaluation is essential to ensure the quality and efficiency of the compiler. However, before starting with the evaluation, we must define and choose benchmarks that represent various cases and select appropriate runtimes for executing the WebAssembly module. So, as there are many options for both benchmarks and runtimes, we need to establish some selection criteria and justifications for our choices.

On the other hand, it is also very important to analyze the build flags and optimization levels offered by TinyGo. These help us improve the performance of the WASM code, so we must check which ones are the most suitable for the .wasm code.

After selecting the algorithms and runtimes, the next step is to start evaluating the performance of this .wasm module. To do so, we will divide this evaluation into three sections:

Q1. Execution Time

First, we start by generating the binaries with the different levels of optimization that TinyGo offers for each benchmark. Then, we run each binary 100 times with the selected runtimes and assess:

1. Which runtime performs the fastest and slowest, and in which contexts or benchmarks.
2. Whether building flags or optimization levels enhance execution speed.
3. The impact of the Garbage Collector on performance.
4. How runtimes perform across the different optimization levels.
5. The identification of any bottlenecks.

Our final goal is to figure out the optimal optimization configuration and runtime for achieving the best execution times, as well as the influence of the garbage collector.

Q2. Binary Size

Additionally, we check which optimizations provide the best reduction in binary size. This is an important feature because smaller binaries enhance portability. Also, TinyGo has been invented to run on small devices by reducing the binary size. So, it is interesting to analyze the percentage reduction.

Q3. System Calls

Finally, we will check the system calls made by each runtime at different levels of optimization. Also, we will calculate different similarity coefficients, such as the Jaccard Similarity Coefficient and the Sørensen-Dice Coefficient, that will help us know which runtimes execute the same system calls, thus being more similar. Therefore, this evaluation is focused on:

1. Identifying which runtime makes the most system calls and which makes the least.
2. Determining which runtimes are most similar to Golang in the sense that they make the same system calls.
3. Analyzing if there is a significant difference in the number of system calls between the optimizations.

4.1.3. Evaluation Environment

In the following Table 4, you can see the environment that we will use to perform the tests (both memory and performance).

Table 4. System environment. *Generated by the author Safia Guellil.*

System overview: Virtual Machine	
Architecture	ARM64
Operating System	Ubuntu 22.04
CPUs	2-4
Go installed version	go1.20.3
TinyGo installed version	0.27.0
Wasmtime installed version	wasmtime-cli 8.0.0
Wasmedge installed version	wasmedge version 0.12.0
Wasmer installed version	4.1.0
iwasmi installed version	0.31.0
Wasmi installed version	1.2.2
Wazero installed version	1.3.1

Note. The number of CPUs can be increased.

4.2. Evaluation of Memory Management

As discussed in the background, one of the outstanding features of Golang is its dynamic and automatic memory management system. In the following section, we analyze how this memory management would look like after compiling it to WebAssembly using TinyGo. This analysis comprises the sections described in the "Methodology for assessing memory management", which include:

- **Q1: Analyzing the linear memory layout structure**
- **Q2: Analyzing Escape to the heap behavior.**
- **Q3: Analyzing Garbage Collector**.

4.2.1. Q1: Analyzing the linear memory layout structure

In this section, we analyze in depth the memory layout of WASM once compiled with TinyGo. Since the memory layout of WASM depends on the compiler [31], we will check that the linear memory corresponds to TinyGo's compiler, as explained previously in the background.

To perform the analysis, we first define which sections of the memory we want to observe: the heap, the stack, and the data. For each of these sections, we aim to determine their starting position and growth behavior, whether it's increasing, decreasing, or random.

Note
We can't show the addresses of stacks with the "print" function because when we require the variable address with the & symbol, it will then escape to the heap. We have tried different solutions, but in the end, we found that the best way to retrieve information related to the memory layout is by modifying some internal functions.

So, we have modified the following internal file: **/usr/local/lib/tinygo/src/runtime/ gc_blocks.go**
We set the garbage collector debug global variable to true and inserted some prints into the *calculateHeapAddresses's* function, as you can see in the following code:

```
/usr/local/lib/tinygo/src/runtime/gc_blocks.go
const gcDebug = true
…
…
func calculateHeapAddresses() {
        …
         if gcDebug {
                println("heap Start:         ", heapStart)
                println("heap End:           ", heapEnd)
                println("total size:        ", totalSize)å
                println("metadata size:     ", metadataSize)
                println("metadataStart:     ", metadataStart)
                println("global Start: ", globalsStart)
                println("global End: ", globalsEnd)
                println("stack Start: ", stackTop)
                println("current stack: ", getCurrentStackPointer())
         }
         …
}
```

After setting up the debugger, we have generated the following simple code to perform some testing. This code calls a function called layout that returns a string. In the main function, this string is printed as well as its pointer. We can see with the go run command that the file works as expected.

```
memory.go
1   package main
2
3   //go:wasm-module main
4   //export layout
5   func layout() (string){
6         text := "Hello world!"
7         return text
8   }
9
10  func main() {
11        text := layout()
12        println(text)
13        println(&text)
14  }
```

```
Go run memory.go
Hello world!
0x400003a758
```

Then, we compile it with TinyGo. We added the **-print-allocs** flag to see if some variable escapes to the heap.

```
tinygo build -target=wasi -o output.wasm -print-allocs=main memory.go

.. /memory.go:11:2: object allocated on the heap: escapes at line 13
```

```
wasmtime run output.wasm
heap Start:           0x000103f0 (66544)
heap End:             0x00020000 (131072)
total size:              0x0000fc10
metadata size:     0x000003e1
metadata Start:     0x0001fc1f (130079)
global Start:        0x00010000 (65536)
global End:          0x000103f0 (66544)
stack Start:           0x00010000 (65536)
current stack:       0x0000ffc0 (65472)
-> found memory heap: 0x000103f0 48 (66544)
-> found memory heap: 0x00010420 16384 (66592)
-> found memory heap: 0x00014420 8 (82976)
Hello world!
0x00014420 (82976)
```

With the output obtained, we can draw several conclusions:

- First, we can see that the linear memory layout is organized as follows, starting at index 0: stack, global, heap, and heap-metadata. So, the stack is at the bottom, the heap is at the top, and in between we have the global data section.
- In the case of the stack, it grows by decrementing the index. That is because the current pointer of the stack, its value, is lower than the starting value.
- Finally, we can see that the variable text has been escaped to the heap because we have tried to obtain its pointer. This fact is evidenced by the "*found memory heap*" debugging text, where the last entry has the same value as the terminal output. Also, the address retrieved is in the range of the heap, as expected.

With this information, we can draw a diagram of Web Assembly's linear memory with its different sections (see Figure 27). We see that the organization of the sections is similar to the way TinyGo organizes the memory. This evidence shows that the compiled WASM linear memory is structured by the TinyGo compiler according to its internal structure.

On the other hand, if we compare the way in which the TinyGo compiler organizes the memory layout with other compilers that cover WebAssembly (see Figure 26), we see that its structure is just like the compiler **clang9**.

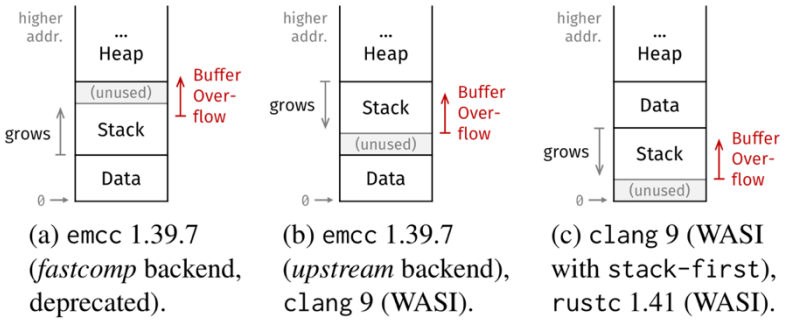

Figure 26. WASM linear memory for different compilers.
https://www.usenix.org/system/files/sec20-lehmann.pdf

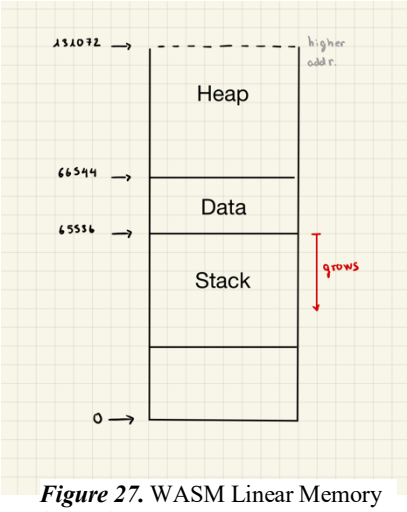

Figure 27. WASM Linear Memory from TinyGo. *Generated by the author Safia Guellil.*

4.2.2. Q2: Analyzing the Escape to the heap behavior

Once we've introduced the memory layout, let's inspect what happens when a variable escapes to the heap. In this case, we are going to analyze two cases: accessing the memory address of a variable and allocating an object larger than the stack size.

4.2.2.1. Escape to the escape: accessing the memory address of a variable

To perform this test, we have created the following codes: the left one doesn't escape to the heap, but the right one does. These codes are composed of three functions: the "*leaf*" function generates a string, the "*intermediate*" function retrieves the string from the "*leaf*" and generates a new one too, and then the two strings are returned to the main function.

Non-Escape

```
package main

//go:wasm-module main
//export leaf
func leaf() (string) {
        x := "hello"
        return x
}

//export inter
func inter() (string, string){
        x := leaf()
        y := "world!"
        return x,y
}

func main() {
        inter()
}
```

Escape

```
1   package main
2
3   //go:wasm-module main
4   //export leaf
5   func leaf() (* string) {
6           x := "hello"
7           return &x
8   }
9
10  //export inter
11  func inter() (string, string){
12        x := *leaf()
13        y := "world!"
14        return x,y
15  }
16
17  func main() {
18          inter()
19  }
```

```
tinygo build -target=wasi -no-debug -gc=leaking -print-allocs=. -scheduler=none
-o output.wasm  X.go
```

```
../escape.go:6:2: object allocated on
the heap: escapes at line 7
```

Using the command **wasm2wat** with each .wasm file, we can review in depth what is happening in these functions. We can see in the following codes that the code that escapes calls **$runtime.alloc** in the "*leaf*" function, which means that it's reserving memory in the heap. This is a clear example of how this escape to the heap behavior has been carried over in the wasm code.

Non-Escape

```
(func $main.leaf (type 1) (param i32)
    local.get 0
    i32.const 5
    i32.store offset=4
    local.get 0
    i32.const 65701
    i32.store
)
```

Escape

```
(func $main.leaf (type 7) (result i32)
    (local i32)
    i32.const 8
    call $runtime.alloc
    local.tee 0
    i32.const 5
    i32.store offset=4
    local.get 0
    i32.const 65701
    i32.store
    local.get 0
)
```

Finally, we have run the above code 100 times for both versions (the one that escapes and the one that doesn't). We can see of Figure 28, that in general, moving this variable to the heap (looking for a free space and copying the value) has not caused a decrease in performance, which is quite positive.

	Mean	Std.Dev.	Min	Median	Max
real	0.046	0.040	0.017	0.038	0.402
user	0.006	0.004	0.000	0.005	0.015
sys	0.012	0.006	0.000	0.011	0.056

Figure 29. 100 execution time of non-Escape code. *Generated by the author Safia Guellil.*

	Mean	Std.Dev.	Min	Median	Max
real	0.045	0.024	0.021	0.038	0.226
user	0.006	0.004	0.000	0.005	0.018
sys	0.012	0.005	0.000	0.012	0.025

Figure 28. 100 execution time of Escape code. *Generated by the author Safia Guellil.*

4.2.2.2. Escape to the heap: allocating an object larger than the available stack

Finally, the second option we will evaluate is the case of allocating an object larger than the available stack.

In the following code, we can see how we reserve an object with 20000 integers (20000·4 = 80000 bytes). Unless otherwise specified, the default stack size is 256 bytes, so we exceed this amount. We see then how the object escapes to the heap, since it is too large to be stored in the stack.

```
escape.go
package main
//go:wasm-module main
//export escapesToHeap
func escapesToHeap ()  {
        s := make([]int, 20000)
        s[0] = 1
}

func main() {
        escapesToHeap()
}
tinygo build -target=wasi -print-allocs=. -o output.wasm escape.go
…/escape.go:6:11: object allocated on the heap: object size 80000 exceeds maximum
stack allocation size 256
```

On the other hand, as for the execution time, we see that it is quite low compared to the previous case, despite allocating a larger object. This could be due to the fact that the object escapes when the make function is executed, i.e., it is not stored in the stack and then moved to the heap, since it escapes at line 6, which corresponds to the "make" function.

	Mean	Std.Dev.	Min	Median	Max
real	0.053	0.062	0.017	0.041	0.642
user	0.006	0.004	0.000	0.005	0.016
sys	0.012	0.004	0.004	0.012	0.025

Figure 30. 100 execution time of Escape code (allocating large object on stack). *Generated by the author Safia Guellil.*

4.2.3. Q3: Analyzing the Garbage Collector

As we have seen in the introduction, one of the most outstanding features that Golang has is its memory management system through the garbage collector. TinyGo also has a GC but is simpler. However, Web Assembly does not have GC, so the compiler adds its memory management system. In the following test, we analyze:

1. Determine if TinyGo's garbage collector is added to the Web Assembly binary.
2. Compare the number of automatic garbage collector cycles between Golang and TinyGo, considering the different TinyGo's GC options.
3. Manually trigger the garbage collector and measure the time spent on one cycle

4.2.3.1. Check if the TinyGo's GC is added in the Web Assembly binary

To verify if GC is added to the .wasm module and is running, we allocate (on the heap) a lot of memory and see if the GC runs automatically. In this case, we are allocating around 80,000 bytes as you can see in the following code.

```
memory.go
package main

import "runtime"

func escapesToHeap ()  {
	s := make([]int, 20000)
	s[0] = 1
}

func printMemStat(ms runtime.MemStats) {
	runtime.ReadMemStats(&ms)
	println("ALL Memory: \t\t",ms.Sys, " bytes")
	println("ALL Memory Heap: \t",ms.HeapSys, " bytes")
	println("Heap Idle: \t\t",ms.HeapIdle, " bytes")
	println("Heap Inuse: \t\t",ms.HeapInuse, " bytes")
	println("Heap objects: ", ms.Mallocs)
	println("Heap objects [freed]: ", ms.Frees)
	println("Heap objects [not-freed]:", ms.Mallocs-ms.Frees)
}

func main() {
	ms := runtime.MemStats{}

	println("################################################")
	println("[Beginning] ")
	printMemStat(ms)

	println("################################################")
	println("[Allocate]")
	escapesToHeap()
	printMemStat(ms)
}
tinygo build -target=wasi -print-allocs=. -o output.wasm memory.go
…/memory.go:8:11: object allocated on the heap: object size 80000 exceeds maximum stack allocation size 256
```

In the output of this execution, we can see how effectively the object has been allocated on the heap because it exceeds the stack size.

Once the code has been compiled and verified that the object is in the heap, it is time to execute this module .wasm. For this, we will use the wasmtime runtime.

```
wasmtime run output.wasm
-> found memory heap: 0x00010410 48
-> found memory heap: 0x00010440 16384
##############################################
[Beginning]
ALL Memory:          64496  bytes
ALL Memory Heap:           63488  bytes
Heap Idle:                 47056  bytes
Heap Inuse:          16432  bytes
Heap objects:  2
Heap objects [freed]:  0
Heap objects [not-freed]: 2
##############################################
[Allocate]
running collection cycle...
-> found memory heap: 0x00014440 80000
ALL Memory:          195568  bytes
ALL Memory Heap:           192544  bytes
Heap Idle:                 96112  bytes
Heap Inuse:          96432  bytes
Heap objects:  3
Heap objects [freed]:  0
Heap objects [not-freed]: 3
```

As we can see in the output, a GC cycle has been performed before increasing the memory size. This fact is evidenced by the message "*running collection cycle...*" that appears before "*found memory heap*", which means that a garbage collector cycle has been executed before starting to search space in the heap for the object in question.

On the other hand, if we take a look at the .wasm code compiled with the wasm2wat tool, we can see that the garbage collector functions of TinyGo have been compiled and implemented. This gives the WebAssembly code a garbage collector, although the execution times are decided by the compiler.

```
wasm-objdump -xh output.wasm
 - func[17] size=1594 <runtime.alloc>
 - func[18] size=310  <runtime.markRoots>
 - func[19] size=30   <(runtime.gcBlock).state>
 - func[20] size=48   <(runtime.gcBlock).markFree>
 - func[21] size=76   <runtime.growHeap>
 - func[22] size=735  <runtime.startMark>
 - func[23] size=88   <runtime.printspace>
 - func[24] size=440  <runtime.printuint64>
 - func[25] size=45   <(runtime.gcBlock).setState>
 - func[26] size=57   <runtime.calculateHeapAddresses>
 - func[27] size=40   <(runtime.gcBlock).findHead>
 - func[28] size=432  <malloc>
 - func[29] size=213  <runtime.hashmapBinarySet>
 - func[30] size=337  <free>
```

In conclusion, we have been able to demonstrate how the WebAssembly functions incorporate and execute the TinyGo garbage collector functions, thus allowing a level of memory management in the .wasm module.

In fact, in the following Figure 31, we can see the call graph that shows the relationship between the functions executed in the .wasm binary of the previous code. These comprise the different functions of the garbage collector cycle (such as *runGC*, *markRoots*, *markStart* and so on) described above in the TinyGo memory management section.

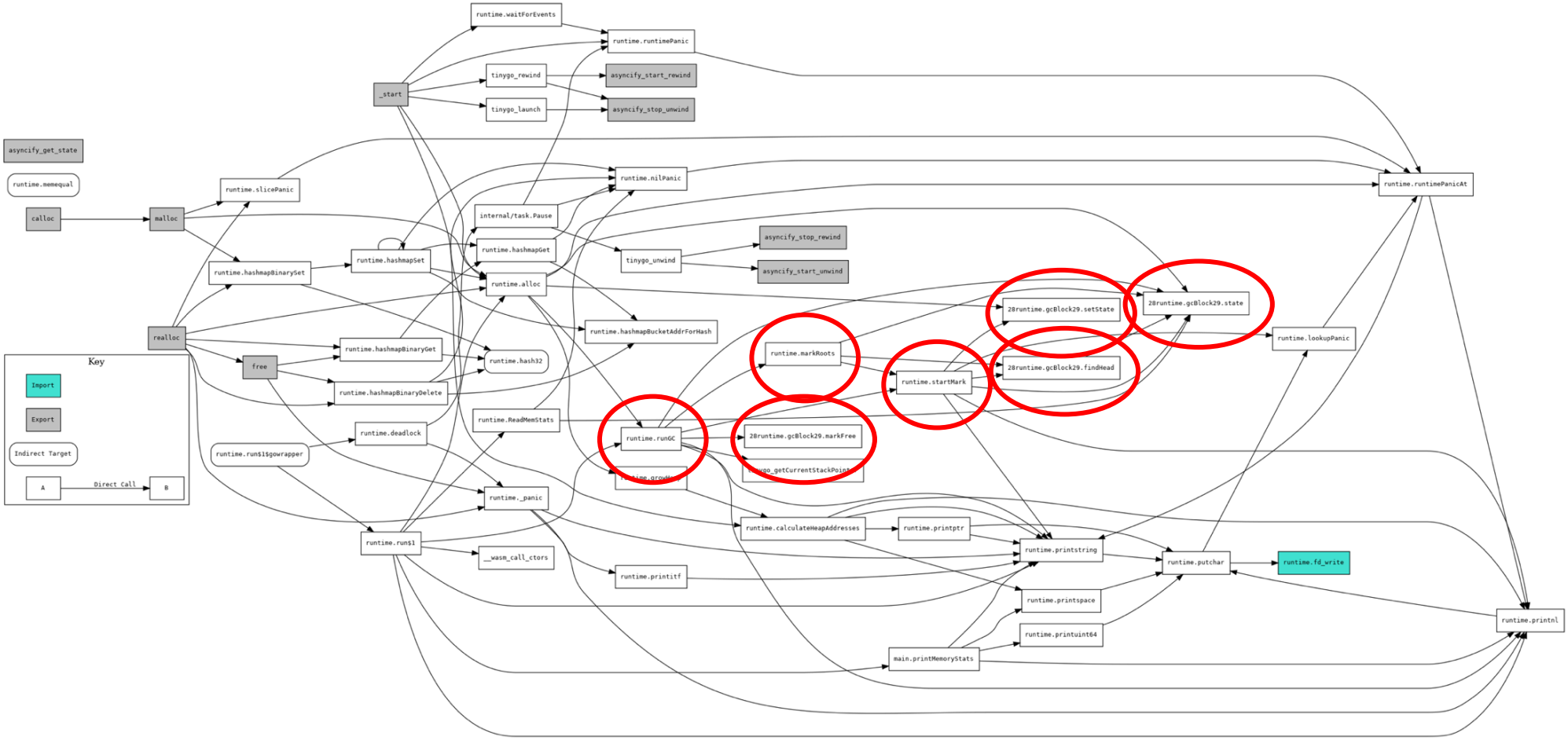

Figure 31. Call Graph of the execution on a wasm module compiled with TinyGo. *Generated by the author Safia Guellil.*

4.2.3.2. Number of automatic GC cycles between Golang and Tinygo

As Garbage Collector is an automatic memory management system, in this test we analyze this feature by counting the number of cycles between Go's GC and TinyGo's GC, the number of freed objects, the size of the binaries, and finally the execution time. In addition, we will study the different garbage collector options offered by TinyGo. These are mainly conservative and precise, but there is also another version that does not free memory, a leaking garbage collector.

Number of Cycles

As we can see in Figure 32, in general, the TinyGo garbage collector performs more cycles than Go's despite the fact that both release the same amount of objects as we observe in Figure 33. This is due to the fact that its algorithm is simpler and more conservative, which generates many false positives that prevent it from correctly freeing the memory.

On the other hand, we see that there is not a big difference in the number of cycles between the accurate and conservative versions of the TinyGo's garbage collector. In theory, the accurate version generates fewer false positives and scans objects better. However, we see that in this case it seems to run a little bit more cycles, which could be related to this more exhaustive object scanning.

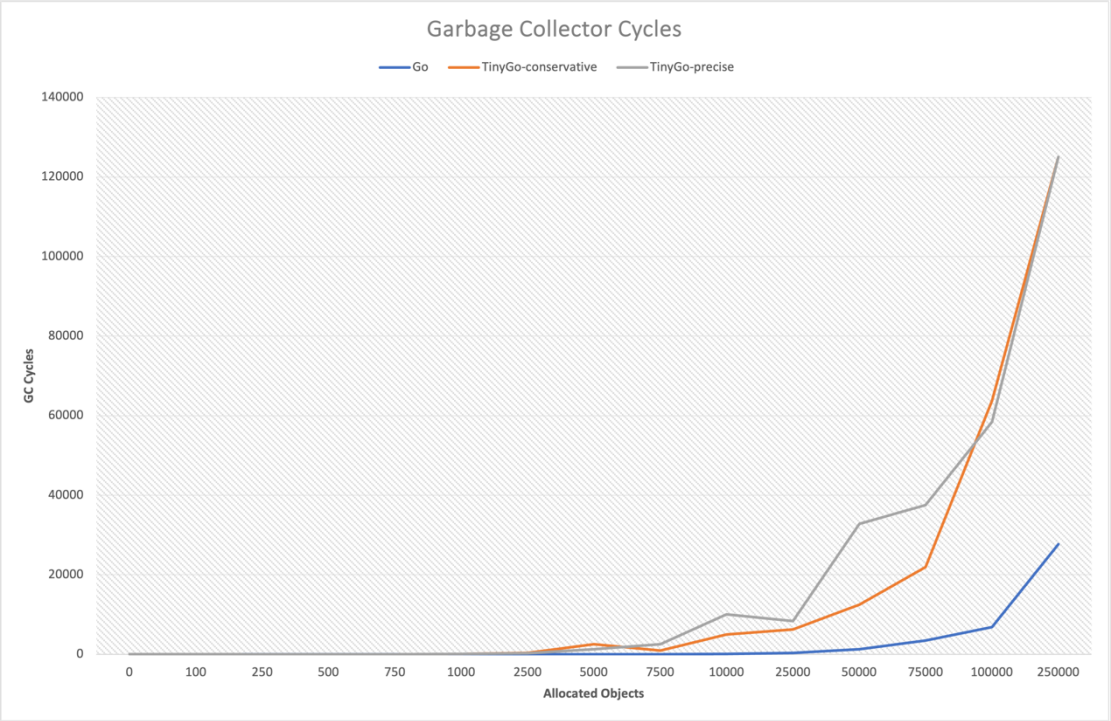

Figure 32. Number of Garbage Collector cycles. *Generated by the author Safia Guellil.*

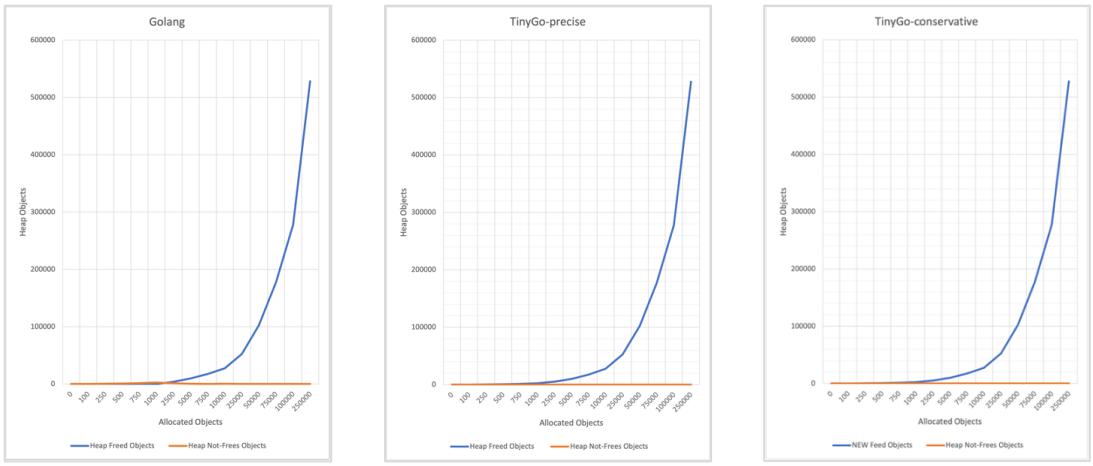

Figure 33. Number of freed objects. *Generated by the author Safia Guellil.*

Size Reduction

On the other hand, as for the size, we see in table X how the binary has been reduced to more than 90% thanks to the TinyGo compiler. This makes the code much more portable.

We can also observe that among the versions offered by TinyGo, the leaking one is the lightest since it does not free the memory allocated in the heap. However, the difference in size is not very significant.

On the other hand, the precise one is a bit heavier than the conservative one, since the garbage collector algorithm has a better object scanner. Again, the difference in size is not very significant.

Table 5. Comparison of binary sizes. *Generated by the author Safia Guellil.*

	Go	compiled Golang	conservative TinyGo	precise TinyGo	leaking TinyGo
size (kilobytes)	1.2	1331.2	89.1	90.6	70.3
Reduction			93%	93%	95%

Execution Time

Finally, in terms of execution time, we observe that the Golang code is much faster than the one compiled with TinyGo. However, the gc-leaking version is faster. This could mean that the execution of the garbage collector cycles of TinyGo is what increases the execution time of the binaries. Also, remember that TinyGo's GC performs many more cycles than Golang's GC, which increases the execution time.

Table 6. Comparison of execution times. *Generated by the author Safia Guellil.*

	compiled Golang	conservative TinyGo	precise TinyGo	leaking TinyGo
time (seconds)	4.58	1202.96	636.52	0.026

4.2.3.3. Trigger manually GC and see how much time spends.

Finally, we have previously calculated the execution time of several garbage collector cycles. In this test, we want to analyze how long a single cycle takes, both in golang and TinyGo. To do this, we will manually run a GC cycle and check the execution time and size of the binary.
The code used for this test is as follows:

```
wasmtime run output.wasm
package main

import (
        "runtime"
        "time"
       )

func main() {

       escapesToHeap(10000)
       start := time.Now()
       runtime.GC()
       elapsed := time.Since(start)
       println("Function execution time: ", elapsed)
}

func escapesToHeap(n int) {
       s := make([]byte, n)
       for i := 0; i < n; i++ {
              _ = append(s, 55)
       }
}
```

Note. It is important to note that when executing this code in TinyGo we observe that internally more than one cycle of the GC is executed, i.e., not only the one specified by us.

Execution Time

In the following Table 7, we show the results obtained by running a single cycle of the garbage collector. We can see that the time is very small in nanoseconds. In general, although the Golang code is faster, there is no significant difference in the execution time with TinyGo. So, we conclude that the execution times obtained in the last test are not because the GC is slower but because this algorithm is executed several times. However, despite being a simpler algorithm, it is a little slower than Golang.

Table 7. Execution time of one GC cycle. *Generated by the author Safia Guellil.*

	compiled Golang	conservative TinyGo	precise TinyGo	leaking TinyGo
time (ns)	179170	230796	205087	2375

Size Reduction

Finally, as in the previous test, we observe that the size of the binary is reduced by more than 90% when using the TinyGo compiler.

Table 8. Comparison of binary sizes. *Generated by the author Safia Guellil.*

	Go	compiled Golang	conservative TinyGo	precise TinyGo	leaking TinyGo
size (kilobytes)	1.2	1433.6	102.6	104.3	86.4
Reduction			92%	92%	94%

4.2.4. Conclusions of the memory management evaluation

Finally, after evaluating the different aspects of the memory system, we present the following conclusions and main insights regarding the WebAssembly linear memory structure, the escape to the heap behavior, and the garbage collector.

4.2.4.1. Conclusions WASM Linear Memory Layout

The linear memory structure of WebAssembly is defined by the compiler [17]. In this test, we have verified that the organization of the sections stack, heap, and data in the .wasm linear memory corresponds to those defined by the compiler TinyGo. Besides, the stack growth is towards lower addresses, which makes the general structure of the WebAssembly linear memory very similar to the one obtained by the compiler clang 9. Therefore, the TinyGo compiler and clang 9 organize WebAssembly's linear memory in the same way.

4.2.4.2. Conclusions Escape to the Heap Behavior

As for the escape to the heap behavior, we have checked two typical cases in which the code should escape: when trying to access the memory address of a variable, and when the size of the object exceeds the size of the stack. Although, as we have seen, the TinyGo compiler does not always escape in practice [26], in these two cases it has escaped correctly.

On the other hand, we see that this behavior is also present in the .wasm module. In fact, we have executed a code that escapes and another that does not, and we could observe (in the .wat file) how in the code that does escape, the *$runtime.alloc* function is executed to reserve memory in the heap to allocate this escaped object. This confirms that this behavior has been transferred to WebAssembly.

Finally, in terms of execution time, this behavior does not add significant additional time to the code.

4.2.4.3. Conclusions Garbage Collector

Lastly, we have analyzed TinyGo's garbage collector, compared it with Golang's GC, and checked if it also runs in the WebAssembly module.

First, we have found that the garbage collector functions of TinyGo have also been compiled and added to the WebAssembly module, which provides the binary with this memory management system. Although, the GC executions are predefined by the TinyGo compiler.

On the other hand, when comparing TinyGo's GC with Golang's, we see that it executes many more cycles despite releasing the same number of objects from the heap. This is basically due to the fact that it is a less complex and conservative garbage collector that often fails to detect reachable objects. Therefore, it performs less heap cleaning and executes more cycles.

In terms of execution time, we see that there is hardly any difference in the time it takes to execute only one GC cycle between TinyGo and Golang. However, because TinyGo executes many more cycles, the final execution time is much longer. On the other hand, in terms of size, TinyGo manages to reduce up to more than 90% of the compiled code of Go.

Finally, regarding the different types of garbage collector offered by TinyGo (conservative and precise), we see that there is no significant difference between the number of cycles, execution time, or binary size. However, the version that does not free the objects and only allocates them to the heap (gc-leaking) is much faster, as it does not incorporate the full GC algorithm.

4.3. Evaluation of Performance

In the next section, we present the performance evaluation in terms of execution time, binary size, and system calls. But first, we must select a set of benchmarks and runtimes to be able to execute the binaries generated with TinyGo and evaluate their performance.

4.3.1. Suite of Benchmarks

The selection of benchmarks is a very important task, as it helps us evaluate different aspects of the generated binary code. In Table 4, you can see the selected algorithms. We have tried to choose different types of algorithms in order to test different libraries, input/output, computational capacity, complexity, size, and so on.

Table 9. Suite of Benchmarks. *Generated by the author Safia Guellil*.

	Algorithm	Category	General Description	Source
1	Quick Sort	Sorting algorithm	Quick sort is a sorting algorithm that is based on the "divide and conquer" technique.	Rosetta code[6]
2	Floyd-Warshall	Graph algorithm	Floyd-Warshall is an algorithm used to discover the shortest paths between all pairs of vertices in a graph with weighted edges.	Rosetta code[7]
3	FASTA	Sequence alignment algorithm	FASTA is an algorithm used to compare a query sequence against a DNA sequence database, aiding in identifying similarities and relationships within genetic data.	Translated to Go from github: greensoftwarelab[8]
4	AES256	Cryptographic algorithm	A simple code that reads a text string from the terminal, encodes it, and decodes it.	github: aziza-kasenova[9]
5	K-NN	Machine Learning	K nearest neighbor is a non-parametric classifier for supervised learning.	github: mattn[10]
6	File System	Process Large Files	A simple code that reads and processes files of different sizes.	github: snassr[11]
7	Parallel Matrix Multiplication	Concurrency & Parallelism	It is a simple matrix multiplication algorithm, but with parallelism.	Implemented by author
8	Network Communication	Sockets	A simple network communication code that attempts to create a TCP listener.	Implemented by author

Important.
The last two algorithms (parallel matrix multiplication and network communication) will be evaluated separately from the rest of the benchmarks. These have been included to evaluate the parallelism, and socket functionalities, not their performance.

[6] https://rosettacode.org/wiki/Sorting_algorithms/Quicksort#Go
[7] https://rosettacode.org/wiki/Floyd-Warshall_algorithm
[8] https://github.com/greensoftwarelab/WasmBenchmarks/tree/master/Benchmarks/Fasta
[9] https://gist.github.com/aziza-kasenova/3aea2160cbaebc5a4ba1b9219cba612e#file-aes256-go
[10] https://github.com/mattn/go-knn-iris/tree/master
[11] https://github.com/snassr/blog-0010-processinglargefilesingo

4.3.1.1. Description of each benchmark

In the following section, we present a more extensive description of how each algorithm works, the main Go libraries it uses, if we have made any changes, and why it has been selected to be part of the benchmark suite.

General Considerations

Before going in depth into each benchmark, we have to describe some important facts that have been applied to all of them:

1- We have avoided using the "*fmt*" library for printing (e.g., "*Fmt.Println()*"), because it requires a lot of supporting functions that increase the binary size [44].
2- To evaluate timing, we are going to consider 100 executions so that we can observe the general trend of the runtime.
3- It's important not to modify the benchmark to make it suitable if the library is not supported by Tinygo.
4- We will have to grant permissions "of capacities" to all the runtimes when they have to access some file in the terminal. To do so, we have to indicate in the runtime command the directory of work.

Quicksort

This sorting algorithm has been obtained from the Rosetta Code page, a wiki that groups different programming algorithms in different languages. The code is quite simple and sorts a small list of 10 integers. It is worth mentioning that the code does not use any external libraries. Also, as mentioned above the "*fmt*" library has been replaced by "*println*" function.

Floyd-Warshall

This algorithm has also been obtained from Rosetta Code as Quick Sort, and it also replaces the "*fmt*" library. In this algorithm, different types of registers or interfaces are implemented, as in the case of the graph interface that you can see below in Figure 34. This feature is quite interesting since, TinyGo interprets the interfaces differently from Golang [15].

Finally, this algorithm creates a small 4-vertex graph with 5 artists of different weights and uses only the "*strconv*" library to convert an integer to a text string.

```
// A Graph is the interface implemented by graphs that
// this algorithm can run on.
type Graph interface {
  Vertices() []Vertex
  Neighbors(v Vertex) []Vertex
  Weight(u, v Vertex) int
}
```

Figure 34. Graph interface of Floyd-Warshall algorithm.
https://rosettacode.org/wiki/Floyd-Warshall_algorithm

FASTA

This algorithm has been obtained from the repository of a research group called *Green Software Lab*[12]. However, because it is written in C, we have used our knowledge to translate it to Golang in order to compile it with TinyGo.

The code stands out mainly for declaring several types of constants and global variables, among them arrays of different sizes and text strings. In addition, it implements different basic functions, such as generating random numbers or dealing with buffers.

We will run the algorithm with three different datasets to analyze the performance of the runtimes as the amount of data to be analyzed increases. These are: "*SMALL_DATASET = 6250000*", "*MEDIUM_DATASET = 12500000*" and "*LARGE_DATASET = 25000000*".

Finally, it is important to note that it uses two libraries: "*os*" to write to the terminal and "*strconv*" to convert an integer to a string.

AES256

This algorithm has been obtained from *Aziza Kasenova*[13] a senior engineer working for Insider Enterprise. This algorithm encrypts and decrypts a text string in 256-bit AES (Advanced Encryption Standard) and in CBC (cipher-block chaining) mode.

We have added a small modification to this algorithm, which is the reading of the message to be encrypted through the terminal. In this way, we will also test the reading or input by the user.

Finally, it should be noted that it is one of the benchmarks that most libraries use: "*bytes*", "*crypto/aes*", "*crypto/cipher*", "*encoding/base64*", and "*os*". Mainly we see that they are libraries to deal with bytes or to be able to implement the AES algorithm. In addition to the "*os*" library that helps us read the message by terminal.

KNN

This algorithm is the K nearest neighbor, an unsupervised classification machine learning algorithm obtained from mattn[14] a long-time Golang user and contributor.

The algorithm uses the iris.csv[15] dataset, which is widely used in the training and testing of self-learning algorithms. This dataset consists of a total of 150 rows.

On the other hand, internally, the algorithm uses Euclidean distance, k=8, and the following libraries: "*encoding/csv*", "*math*", "*os*", "*sort*", and "*strconv*".

Finally, a struct is also defined, as you can see in Figure 35. This struct is formed by the “k” value, the attributes "XX" and the classification "Y".

```
type KNN struct {
        k  int
        XX [][]float64
        Y  []string
}
```

Figure 35. Struct in KNN algorithm. *https://github.com/mattn/go-knn-iris/tree/master*

[12] https://greenlab.di.uminho.pt/
[13] https://www.linkedin.com/in/aziza-kasenova/?originalSubdomain=tr
[14] https://github.com/mattn
[15] https://github.com/mattn/go-knn-iris/blob/master/iris.csv

File System

The algorithm was obtained from the Medium article *Processing Large Files with Go (Golang)*[16] written by *Snaser*. This article compares two versions of processing large files sequentially and concurrently with Golang. For our benchmark, we have chosen the sequential version, since concurrency will be treated with the parallelism benchmark.

This algorithm reads a file with information related to employees, as you can see in Figure 37. It then processes each line of the file, extracting the necessary information according to the "*result*" interface (Figure 36).

```
type result struct {
        numRows            int
        peopleCount        int
        commonName         string
        commonNameCount    int
        donationMonthFreq  map[string]int
}
```

Figure 36. Struct to show results. https://medium.com/@snassr/processing-large-files-in-go-golang-6ea87effbfe2

The algorithm uses the following libraries: "*bufio*", "*log*", "*os*", and "*strings*". The "*bufio*" library is used to read and write to files.

Finally, it is important to mention that we will run this algorithm with different file sizes or employer information: "*SMALL=40*", "*MEDIUM=4000*" and "*LARGE=40000*". The location and name of the file are entered through the terminal and in the event of not finding it or not being able to open it, the "log" library is used to report the error.

```
C00580100|A|YE|P2020|201903139145682053|15|IND|BALTHASAR,
SUSAN|INDIAN
SHORES|FL|33785|RETIRED|RETIRED|11072018|100||SA17A.8965|1319152|||40
31420191645043402

C00580100|A|YE|P2020|201903139145683412|15|IND|GAZZIER,
JAY|FRIENDSWOOD|TX|77546|MERRILL LYNCH|FINANCIAL
ADVISOR|11072018|100||SA17A.13989|1319152|||4031420191645048289
```

Figure 37. File with employee information. https://medium.com/@snassr/processing-large-files-in-go-golang-6ea87effbfe2

Parallel Matrix Multiplication

This algorithm has been implemented in Go by the authors of this thesis. It is a matrix multiplication in parallel form, following the model of Figure 38. With this algorithm, we will check what happens with parallelism in Web Assembly and TinyGo.

The code uses the following Golang libraries: "*math/rand*", "*os*", "*strconv*", "*runtime*", "*sync*", and "*time*". In this case, we highlight the "*sync*" library to execute synchronization primitives such as WaitGroups to wait for the execution of the different goroutines.

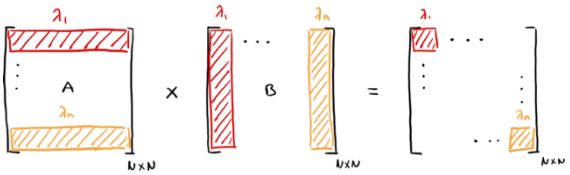

Figure 38. Parallel matrix multiplication. *Generated by the author Safia Guellil.*

Finally, we pass as a parameter the number of threads with which we want to run the code.

Network Communication

Finally, this algorithm will try to create a TCP listener on port 3000 in order to evaluate network and socket communication in TinyGo and Web Assembly. This algorithm has been created by the author of the thesis, and the only library that will be used is "*net*".

[16] https://medium.com/@snassr/processing-large-files-in-go-golang-6ea87effbfe2

4.3.2. Suite of Runtimes

As we have mentioned earlier, Web Assembly doesn't have a runtime, so we must use another tool to run these binary files outside the browser. In the following section, we will show the process of selecting the runtimes to execute the WASM code compiled with TinyGo. For this selection, we did some research and found the *Awesome Web Assembly Runtimes*[17] GitHub repository that lists the different current runtimes.

This list of runtimes has been used to perform the selection according to the following rules of selection criteria:

1. **Open Access**.
2. **Active projects**: the project should be active and in continuous improvement and development. So, the last commit on their git repositories should be this year.
3. **Compatible with TinyGo and Linux**: we should be able to run them correctly on our system.
4. **Production version**: the project should have a production version ready to be used, and the last release published this year.
5. **CLI**: it should have a command line tool to be used on the terminal.

4.3.2.1. Selected Runtimes

Finally, we have selected the following six runtimes from the repository described above that fit the rules established before (Table 10). These runtimes are active projects in continuous development, and we have tested them in our system, so we have verified that they work perfectly.

Table 10. Selected Runtimes. *Generated by the author Safia Guellil.*

#	Runtime	General Description
1	Wasmtime	"*It is a fast and secure runtime for Web Assembly developed by the Bytecode Alliance*[18] *project*". [19]
2	Wasmedge	"*It is a lightweight, high-performance, and extensible web assembly runtime for cloud native, edge, and decentralized applications*." [20]
3	Wasmer	"*Wasmer is a blazing fast and secure Web Assembly runtime that enables incredibly lightweight containers to run anywhere: from Desktop to the Cloud, Edge and even the browser*." [21]
4	iwasm	"*The executable binary built with WAMR (Web Assembly Micro Runtime) VMcore supports WASI and command line interface*." [22]
5	Wasmi	"*It is an efficient Web Assembly interpreter with low-overhead and support for embedded environment such as Web Assembly itself*." [23]
6	Wazero	"*It is the only zero dependency Web Assembly runtime written in Go*." [24]

[17] https://github.com/appcypher/awesome-wasm-runtimes#contents
[18] https://bytecodealliance.org/
[19] https://wasmtime.dev/
[20] https://wasmedge.org/
[21] https://github.com/wasmerio/wasmer
[22] https://github.com/bytecodealliance/wasm-micro-runtime
[23] https://github.com/paritytech/wasmi
[24] https://wazero.io/

4.3.2.2. Discarded Runtimes

On the other hand, at the following Table 11, we can find another list with the discarded runtimes and the justification for the discard.

Table 11. Discarded Runtimes. *Generated by the author Safia Guellil.*

#	Runtimes	Reason for discard
1	*Wasmo*[25]	Last commit was on 2nd April 2022, more than a year.
2	*Existism*[26]	Run WASM code embedded into apps written in different languages.
3	*Fizzy*[27]	When we executed it, we found an error: "*Error: Exception: unknown section encountered 12*". So, it's not compatible with our system.
4	*aWsm*[28]	There have been no more commits since September 22, 2022. Also, there is no release version, which means there is not a production version yet.
5	*EOSIO*[29]	The last commit was in 2019.
6	*Happy New Moon with Report*[30]	It has been more than a year since the last commit.
7	*Lucet*[31]	In 2020, the Lucet team switched focus to working on the Wasmtime engine. This project is not active anymore.
8	*WAKit*[32]	Last commit last year. Also, this project is described as: "Highly experimental. Do not expect to work."
9	*WAVM*[33]	The last commit was a year ago.
10	*TWVM*[34]	"TWVM is still under continuously construction, not production-ready yet!"
11	*SWAM*[35]	Repository archived, last realize in 2020.
12	*Innative*[36]	The last commit was a year ago.
13	*Wasm3*[37]	The last release version more than a year ago.
14	*WasmVM*[38]	The last release was a year ago.

[25] https://github.com/appcypher/wasmo
[26] https://extism.org/
[27] https://github.com/wasmx/fizzy
[28] https://github.com/gwsystems/aWsm
[29] https://github.com/EOSIO/eos-vm
[30] https://github.com/fishjd/HappyNewMoonWithReport
[31] https://github.com/bytecodealliance/lucet
[32] https://github.com/swiftwasm/WAKit
[33] https://github.com/WAVM/WAVM
[34] https://github.com/Becavalier/TWVM
[35] https://github.com/satabin/swam
[36] https://github.com/innative-sdk/innative
[37] https://github.com/wasm3/wasm3
[38] https://github.com/WasmVM/WasmVM

4.3.3. Building Flags

In the following section, we analyze the building flags that the TinyGo compiler offers. This analysis is important since it could affect the performance of the generated binary both at runtime and in terms of size. In the attached Table 12, you can find some interesting building flags obtained from the official TinyGo website [32].

Table 12. TinyGo interesting building flags. *Generated by the author Safia Guellil.*

Type	Flags	Description
Target System	**-target=wasi**	Use the Web Assembly Interface to run the binary outside the browser.
Optimization level	**-opt=0**	Turn off all optimizations.
	-opt=1	Enables only some optimizations, such as disabling the inliner.
	-opt=2	Enables most optimizations and will probably result in the fastest code.
	-opt=s	Similar to -op=2 but providing a balance between performance and size.
	-opt=z	It is the default optimization similar to -op=s but more focused on reducing code size.
Memory Management	**-gc=none**	It does not use any memory management system.
	-gc=leaking	It allocates memory very quickly but never releases it.
	-gc=conservative	Simple and conservative garbage collector, but very unpredictable since any assignment can trigger a garbage collection cycle.
	-gc=precise	It has fewer false positives than the "conservative" version and makes the GC somewhat faster.

4.3.3.1. Recommended Optimizations for Speed

The following are some recommendations from the official documentation of TinyGo to improve the speed [32]:

- **-opt=2** it will probably result in the fastest code.
- **-gc=leaking** sometimes have a great effect on Web Assembly.
- **-scheduler=none** in WASM it might help because WASM does not have native support for stack switching.
- **-panic=trap** it could help by reducing the size of the code.

4.3.3.2. Recommended Optimizations for Size

The following are some recommendations from the official documentation of TinyGo to reduce the binary size [32]:

- **-opt=z** focused on reducing code size.
- Do not import large packages like "*fmt*" use println instead.
- **-no-debug** reduces size by removing debugging symbols.
- **-scheduler=none** disables goroutine support and some code-reducing reaffirmations.
- **-gc=leaking** reduces the code size because it is lighter than the conservative gc.
- **-panic=trap** can reduce the size of the code that corresponds to the panic messages.

4.3.3.3. Final Compilation Code

We have used all optimizations (**O0, O1, O2, OS, OZ**) with the default gc, plus the combination **O2+GC** leaking in the speed tests and the combination **OZ+GC** leaking in the size tests.

```
GOARCH=arm64 GOOS=linux tinygo build -target=wasi -scheduler=none -panic=trap -no-debug -opt=$1 -o ./compiled/benchmark.wasm benchmark.go
```

4.3.4. Q1: Execution Time

Once the set of benchmarks and runtimes have been selected, and the different optimization options have been analyzed, it is time to compile and start running the binary file to perform the performance analysis.

General Overview

In the following **Table 13**, we have collected the best execution times regardless of the optimizations applied. We can observe that, in most cases, the performance is better than that obtained by running the binary in Go. In general, the runtimes with which we have obtained the best results (regardless of optimization) have been wasmtime and wazero, and the worst have been wasmedge and wasmer.

In the case of the wasmedge runtime, its performance is a bit ambiguous. We see that when it comes to low-complex algorithms (Quicksort) or algorithms that do not handle a lot of data (AES256) the performance is very good. However, as we increase the number of the dataset, as in the case of FASTA and File System, we see how it gets worse, obtaining the worst values.

In general, we see that the WASM code compiled with TinyGo obtains good results in most of the selected runtimes, since the execution time is quite comparable to that obtained with Golang. Especially for the cases of wazero, which is a runtime implemented in Go, and wasmtime project whose goals are speed and security.

This was a general overview; in the next section, we will analyze the performance, taking into account the optimizations offered by TinyGo and the garbage collector.

Table 13. Best execution times. *Generated by the author Safia Guellil.*

Best Execution Times	Go	Wasmtime	Wasmedge	Wasmer	iwasm	wasmi	wazero
QuickSort	0.0030	0.0216	0.0000	0.0811	0.0000	0.0000	0.0000
Floyd-Warshall	0.0020	0.0241	0.0000	0.0904	0.0000	0.0000	0.0000
FASTA (SMALL)	0.3440	0.2839	3.8784	0.7942	0.6727	1.7626	0.2726
FASTA (MEDIUM)	0.6380	0.5325	7.2101	1.5413	1.3685	3.3483	0.5691
FASTA (LARGE)	1.2540	1.0789	15.7506	3.1689	2.7512	7.2380	1.1355
AES256	0.0030	0.0229	0.0001	0.0842	0.0000	0.0000	0.0000
KNN	0.0070	0.0370	0.0514	0.0850	0.0100	0.0205	0.0113
File System (SMALL)	0.0040	0.0251	0.0062	0.0932	0.0001	0.0000	0.0004
File System (MEDIUM)	0.0110	0.0417	0.2229	0.1019	0.0410	0.1037	0.0312
File System (LARGE)	0.0700	0.1154	2.3141	0.2552	0.4277	1.1290	0.4280

Optimizations

Regarding optimizations, applying them does improve the execution time in most runtimes, as we can see in Figure 42. In fact, the O0 optimization is the worst performer.

For example, we can observe this improvement with the wasmedge runtime, which, thanks to the application of optimizations, has been able to reduce the time significantly. This can be seen in the case of FASTA (LARGE) which without optimizations takes about 50 seconds (O0), but thanks to the application of optimizations, the time is reduced to 15 seconds.

In Figure 39, we present three graphs in which we can observe this performance improvement in wasmedge thanks to the optimizations.

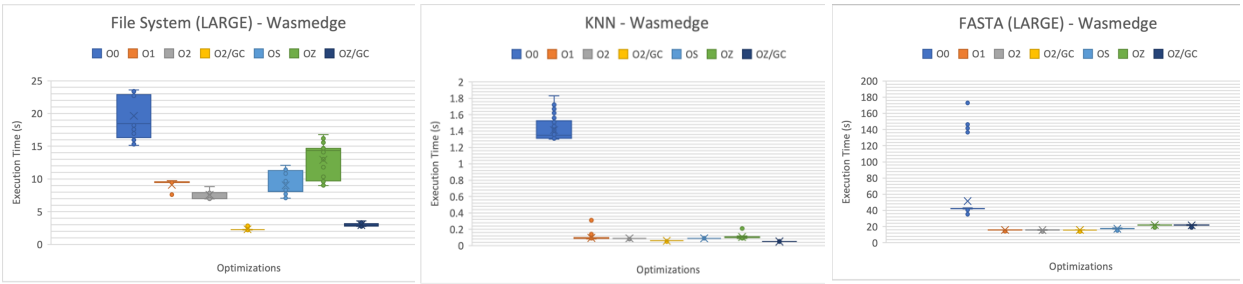

Figure 39. Wasmedge runtime performance. *Generated by the author Safia Guellil.*

In the cases of wasmtime and wazero, which, as we have seen in the general overview, have given the best results, the optimizations do not affect them too much. This is normal and expected since the times offered are quite low compared to the average runtimes. In Figure 40, we see wasmtime and wazero improve very little by applying the optimizations to the most expensive algorithms (FASTA-LARGE and File System LARGE).

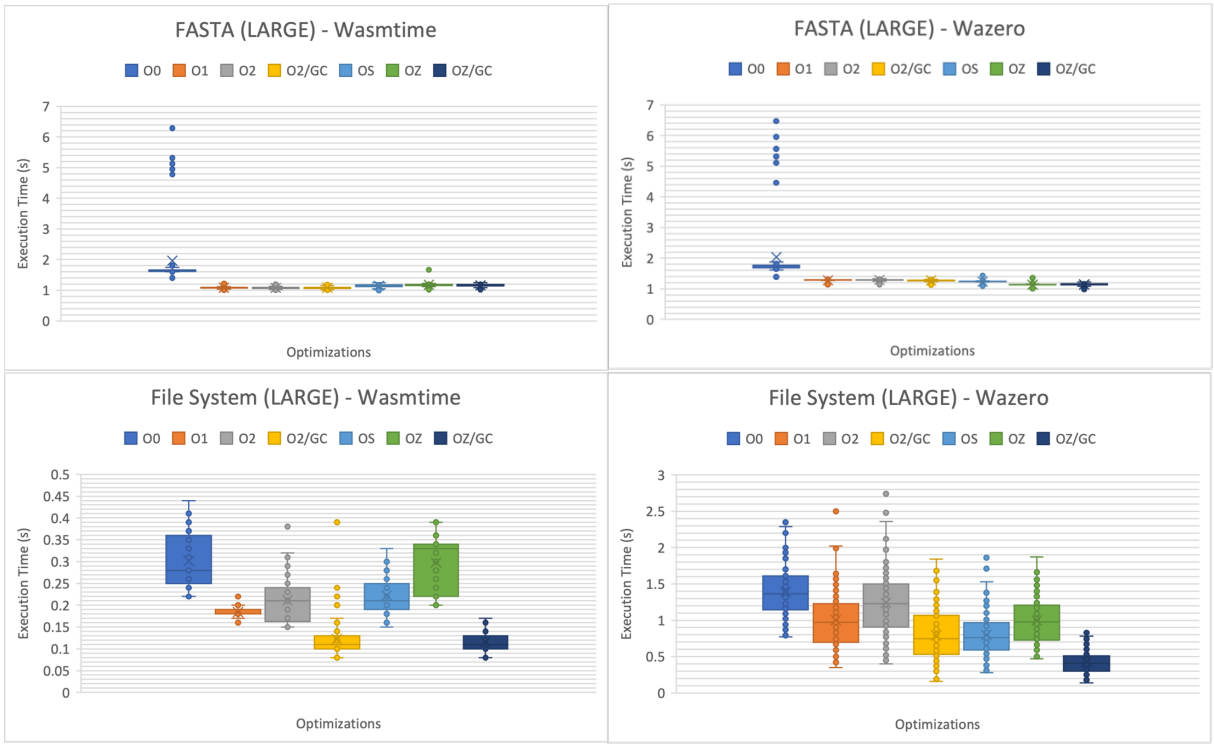

Figure 40. Wasmtime and Wazero performance on File System and FASTA algorithms. *Generated by the author Safia Guellil.*

Finally, regarding the optimizations, the one that generally improves the execution time is the combination of O2 with gc-leaking (O2/non-GC). In fact, this is the optimization recommended by TinyGo to improve performance.

We can see in Figure 41 that in the case of the File System (LARGE) algorithm (the one that generates more execution time), O2/non-GC is the optimization with the best results. However, the OZ/non-GC optimization also gives comparable results, but it is more focused on reducing the binary size.

Note.
In Figure 41, the optimization O2 with gc-leaking is the one in yellow and is shown as “O2/GC” instead of "O2/non-GC" to reduce the legend space.

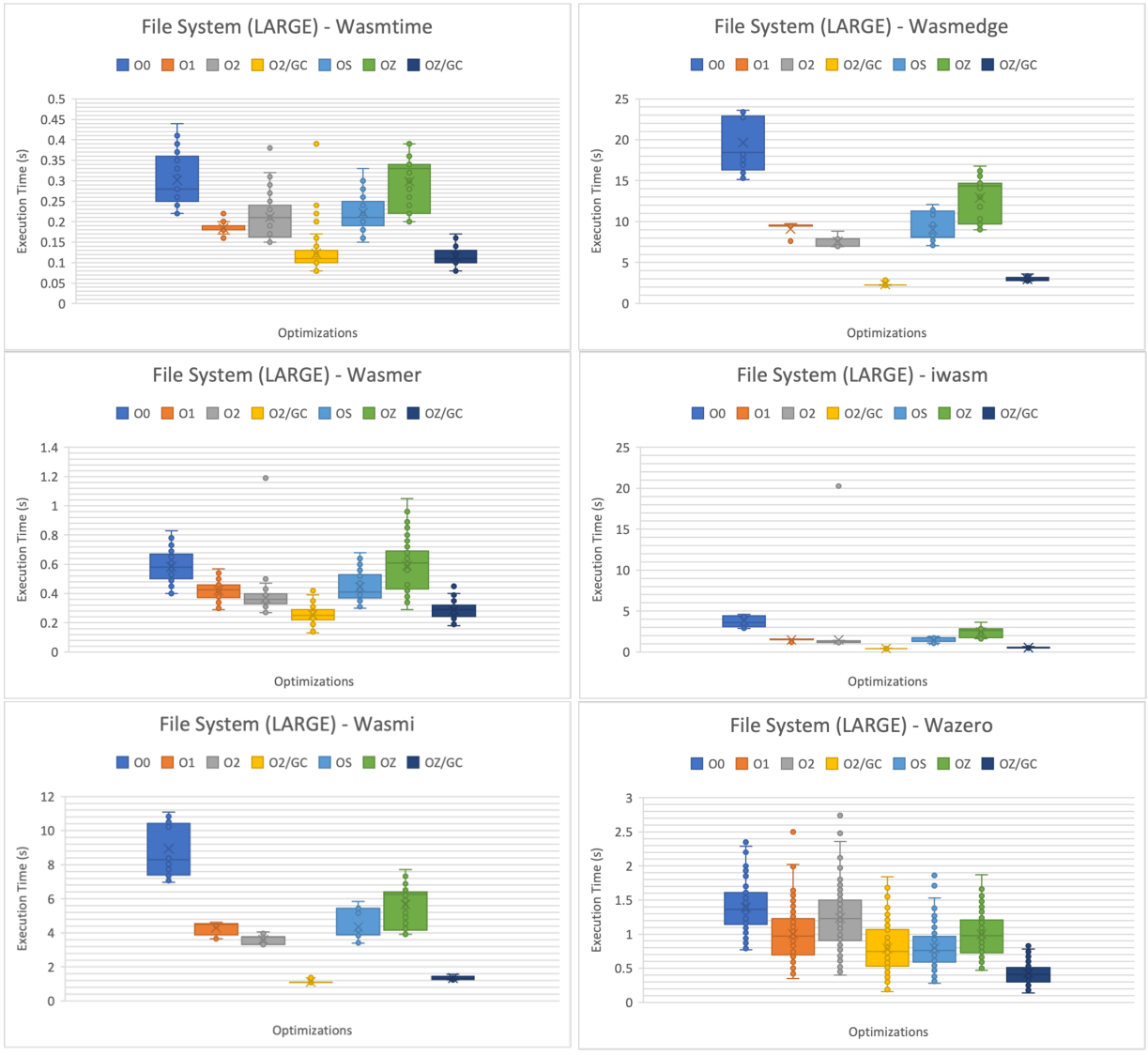

Figure 41. Runtimes performance with the File System (LARGE) algorithm. *Generated by Safia Guellil.*

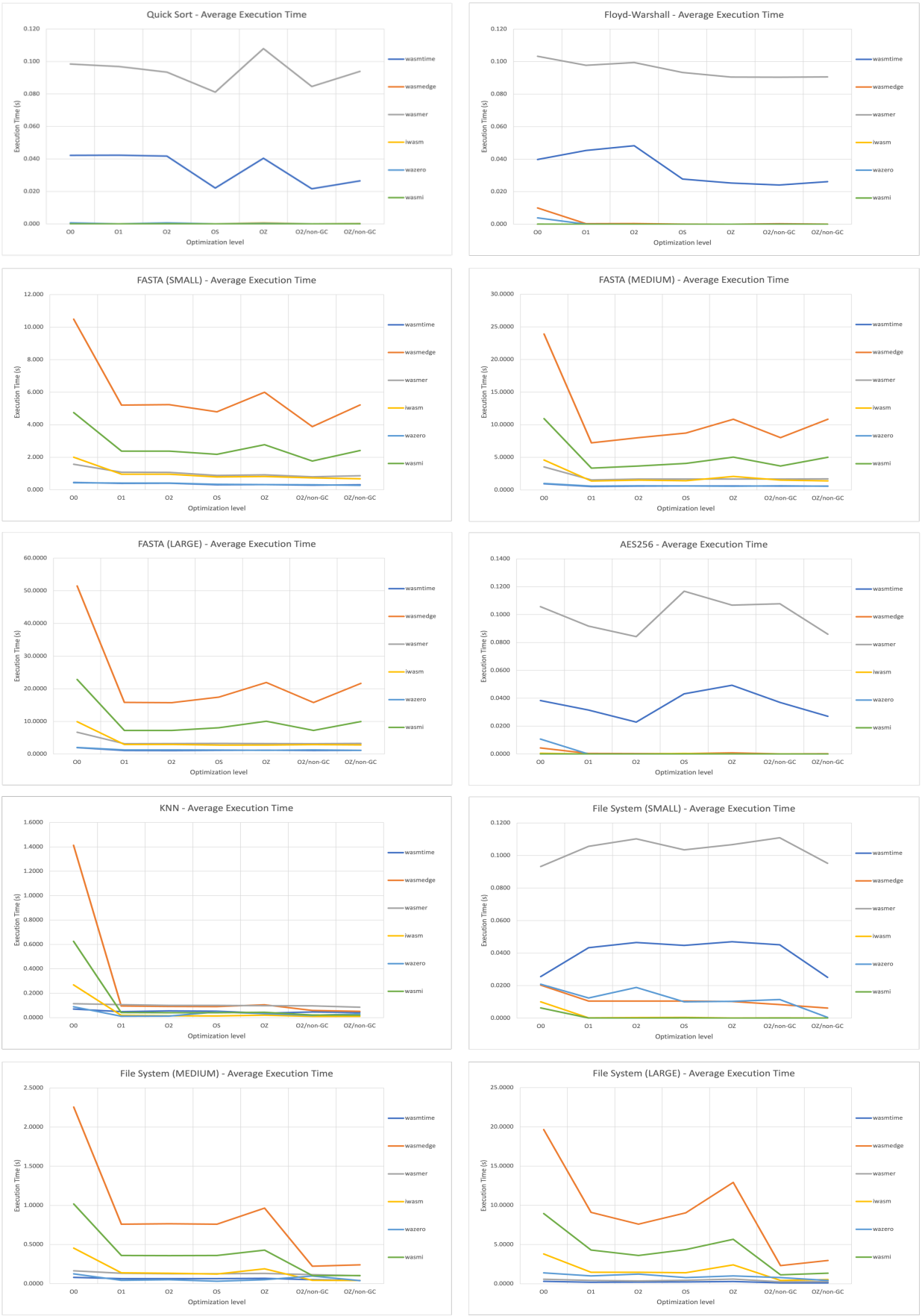

Figure 42. Average of 100 runs with different runtimes and optimization levels. *Generated by the author Safia Guellil.*

Garbage collector

Finally, regarding the garbage collector, in Figure 43, we show two examples in which the O2 and OZ optimizations have been applied with and without the garbage collector. In general, we observe that, depending on the runtime, it affects it in a more or less drastic way. For example, in the case of Wasmedge it is quite clear that the garbage collector code affects the performance because, with the OZ optimization, it goes from 14 to 3 seconds, and with the O2 optimization, it goes from 7 to 2 seconds.

Other runtimes, such as wasmi and iwasm are also affected, but to a much lesser extent, around 1 or 2 seconds. However, the runtimes wasmtime, wasmer and wazero do not seem to be affected by garbage collector cycles at runtime.

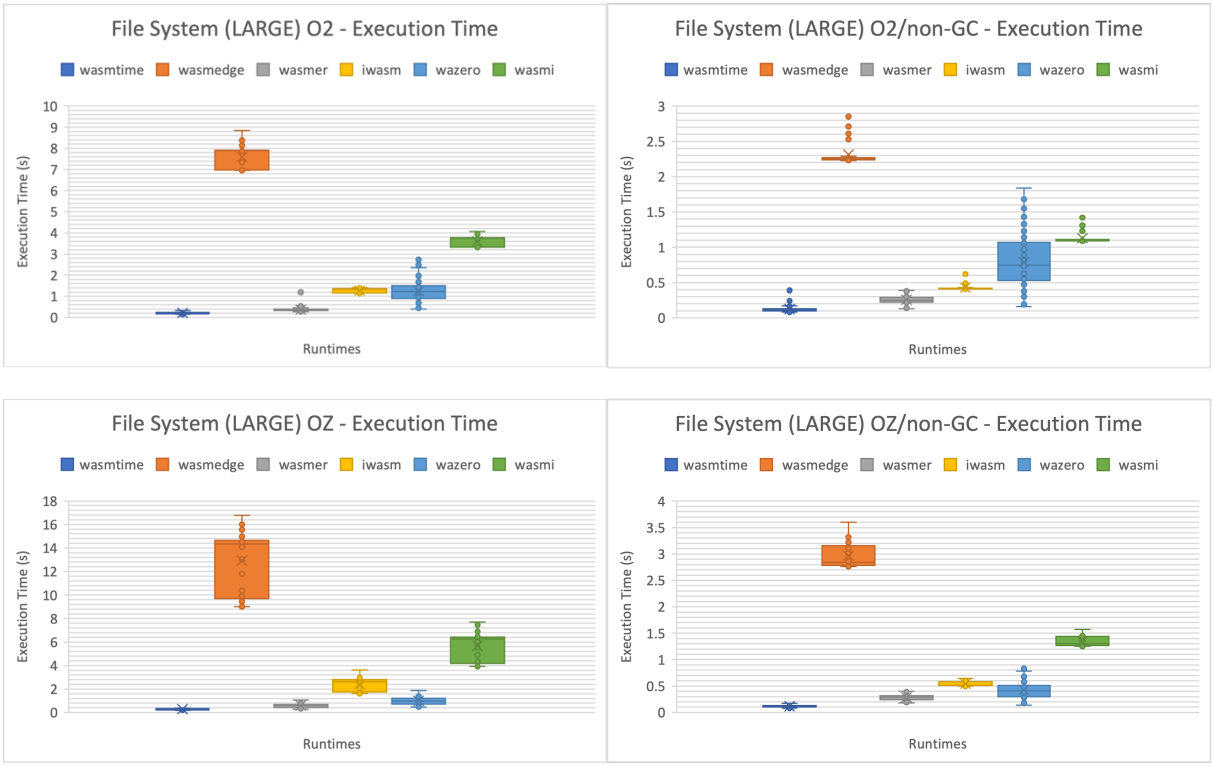

Figure 43. Performance with and without conservative garbage collector. *Generated by the author Safia Guellil.*

4.3.5. Q2: Binary size

This second evaluation is focused on analyzing the size of the binaries generated with the different optimization levels offered by TinyGo.

Table 14 shows the results obtained. We see that in general, even without applying any optimization (as in the case of O0), more than 80% of the binary code is reduced. This is quite favorable, as it shows the efficiency of TinyGo.

On the other hand, the best result with a reduction of more than 99% is obtained with OZ optimization with the gc-leaking. Recall that gc-leaking is a version of the memory management system that reserves memory on the heap but does not free it. Therefore, it is normal that the code is further reduced by not incorporating the garbage collector functions.

However, it is important to note that between the OZ and OZ-nonGC optimizations, there is not a big difference in terms of binary size. Therefore, whether to incorporate the conservative garbage collector does not significantly affect the final size of the binary. Although, it is interesting to apply some optimization, as it can reduce the size of the binary by up to 10%.

Finally, it is interesting to note how some binaries go from weighing 2 megabytes to around 60 kilobytes, as is the case of the KNN algorithm.

Table 14. Binary sizes for each optimization level of TinyGo. *Generated by the author Safia Guellil.*

size (kilobytes)	O0	O1	O2	OS	OZ	O2/non-GC	OZ/non-GC
				Quicksort			
TinyGo	38.00	12.00	11.00	7.80	7.80	7.80	4.60
% Reduction	97.15%	99.10%	99.17%	99.41%	99.41%	99.41%	99.65%
Go Compiled	1331.20						
Go	2.40						
				Floyd-Warshall			
TinyGo	79.00	16.00	24.00	19.00	12.00	11.00	7.40
% Reduction	94.07%	98.80%	98.20%	98.57%	99.10%	99.17%	99.44%
Go Compiled	1331.20						
Go	2.50						
				FASTA			
TinyGo	139.00	13.00	12.00	8.20	11.00	10.00	6.10
% Reduction	90.95%	99.15%	99.22%	99.47%	99.28%	99.35%	99.60%
Go Compiled	1536.00						
Go	3.70						
				AES256			
TinyGo	165.00	36.00	34.00	30.00	28.00	27.00	23.60
% Reduction	89.93%	97.80%	97.92%	98.17%	98.29%	98.35%	98.56%
Go Compiled	1638.40						
Go	2.20						
				KNN			
TinyGo	333.00	91.00	82.00	69.00	68.00	68.00	58.00
% Reduction	83.74%	95.56%	96.00%	96.63%	96.68%	96.68%	97.17%
Go Compiled	2048.00						
Go	2.30						
				File System			
TinyGo	286.00	164.00	143.00	108.00	105.00	122.00	89.00
% Reduction	86.04%	91.99%	93.02%	94.73%	94.87%	94.04%	95.65%
Go Compiled	2048.00						
Go	2.30						

4.3.6. Q3: System Calls

Once the execution speeds and sizes have been analyzed, it is time to look at the system calls. In this section, we analyze the number of system calls made by each runtime with each benchmark, we see if they increase or decrease with the optimization levels, and finally, using similarity metrics, we calculate which runtimes are more similar according to their system calls.

4.3.6.1. Number System Calls

To perform this test, we have used the Linux tool "*strace*".

In the following Table 15, you can see the results of the system calls for each runtime. These results are the lowest (best) ones obtained with the different optimization levels. We can observe that the number of system calls made by the runtimes wasmtime, iwasm and wasmi are the most similar to Go.

On the other hand, wasmedge stands out as the one that makes the fewest system calls, even less than golang. This is quite positive, as it could be linked to a minimal use of system resources.

As for the runtime wasmer, this is the one that seems to get the worst results in general. However, like wasmedge it is the most consistent, i.e., there are no major variations between the different benchmarks in terms of system calls and even between the different optimization levels. This makes it stable in resource usage.

We cannot say the same for the wazero runtime, which, despite being implemented in Golang, offers quite unstable results. In Table 15 (where we show the best results of each runtime), it seems to return normal results, but in reality, if we look at Figure 44, we see that depending on the optimization applied, the result can be quite disastrous. For example, the machine learning algorithm (KNN) with the optimization O1 reaches 300,000 system calls, which is a gigantic number.

Finally, we note that the optimizations (see Figure 44) do not decrease the number of system calls. However, in the runtime, Wazero can improve or worsen its results in a way that seems spontaneous.

Table 15. Best number of system calls (independently of the optimization level). *Generated by the author Safia Guellil.*

System Calls	Go	Wasmtime	Wasmedge	Wasmer	iwasm	wasmi	wazero
QuickSort	179	216	180	1245	95	109	106
Floyd-Warshall	255	226	183	1254	107	120	206
FASTA (SMALL)	9277	9285	183	2312	9163	9176	9309
FASTA (MEDIUM)	18624	18344	183	2326	18222	18235	18415
FASTA (LARGE)	36598	36462	183	2314	36341	36354	235
AES256	174	217	184	1244	96	109	197
KNN	179	224	197	2276	115	123	220
File System (SMALL)	183	223	202	1259	115	120	219
File System (MEDIUM)	336	383	202	2341	272	277	279
File System (LARGE)	816	2206	202	1261	2096	2102	241

Figure 44. Number of System Calls for each benchmark and runtime. *Generated by the author Safia Guellil.*

4.3.6.2. Similarity of System Calls

On the other hand, we also analyze the type of calls made by each runtime to detect which one is the most similar to Golang. For this purpose, we will use the following similarity metrics:

- *Jaccard Similarity Coefficient*: Measures the similarity between two sets based on the size of their intersection and union.
- *Sørensen-Dice Coefficient*: Similar to the Jaccard index but emphasizes shared elements more than the Jaccard index.

On the next pages, you will find tables with the calculated coefficients.

Jaccard Similarity Coefficient

In Table 17, we can observe this coefficient calculated for all the runtimes. We can see that the runtime that executes more similar calls to Golang is Wazero. This result is quite expected since this runtime is written in Go. On the other hand, the least similar is wasmi which is more like iwasm.

As for the rest of the runtimes, we can see that wasmtime and wasmer use about 70% of the same system calls. In addition, we see that the runtime wasmer resembles several other runtimes, such as wasmtime, wasmedge and iwasm. This is due to the fact that wasmer (as well as wasmtime) uses more diverse system functions than the rest of the runtimes, as shown in Table 16 with the quicksort algorithm. Then there are more chances to have more similarities with other sets.

Sørensen-Dice Coefficient

Finally, in Table 18, we can observe the results obtained with the Sørensen-Dice Coefficient. It has similar results to Jaccard Similarity Coefficient, but more conclusive since it focuses more on the elements shared in both sets.

Table 16. System Calls of each runtime with the Quicksort algorithm.
Generated by the author Safia Guellil.

Syscalls Functions Quick Sort						
Go	Wasmtime	Wasmedge	Wasmer	iwasm	wasmi	wazero
clone	fcntl	futex	rt_sigprocmask	mprotect	read	rt_sigaction
write	ioctl	munmap	futex	munmap	ioctl	mmap
futex	mkdirat	brk	clone	mmap	openat	futex
mmap	faccessat	mmap	mprotect	openat	munmap	rt_sigprocmask
rt_sigprocmask	openat	clone	mmap	read	statx	clone
rt_sigreturn	close	rt_sigprocmask	read	newfstatat	getrandom	getdents64
openat	read	close	openat	brk	write	epoll_ctl
close	write	getdents64	brk	close	writev	fcntl
read	writev	newfstatat	ioctl	fcntl	brk	pipe2
sched_getaffinity	ppoll	mprotect	statx	ioctl	mmap	munmap
sigaltstack	readlinkat	rt_sigaction	rt_sigaction	writev	close	openat
rt_sigaction	newfstatat	openat	munmap	sigaltstack	lseek	prlimit64
gettid	set_tid_address	getcwd	getrandom	madvise	sigaltstack	close
execve	futex	fstat	close	prlimit64	faccessat	rt_sigreturn
	set_robust_list	ioctl	eventfd2	rt_sigaction	ppoll	sigaltstack
	clock_getres	faccessat	sigaltstack	set_tid_address	newfstatat	epoll_create1
	sched_getaffinity	read	prlimit64	sched_getaffinity	set_tid_address	gettid
	sigaltstack	readlinkat	sched_getaffinity	getrandom	set_robust_list	read
	rt_sigaction	set_tid_address	epoll_create1	rseq	sched_getaffinity	write
	rt_sigprocmask	set_robust_list	mkdirat	set_robust_list	rt_sigaction	fstat
	getpid	execve	ppoll	faccessat	execve	sched_getaffinity
	brk	prlimit64	epoll_ctl	execve	mprotect	execve
	munmap	getrandom	newfstatat		prlimit64	mprotect
	mremap	rseq	write		rseq	
	clone		lseek			
	execve		fcntl			
	mmap		faccessat			
	mprotect		set_tid_address			
	prlimit64		set_robust_list			
	getrandom		execve			
	memfd_create		rseq			
	membarrier					
	statx					
	rseq					

Table 17. Jaccard Similarity Coefficient. *Generated by the author Safia Guellil.*

	Go	wasmtime	wasmedge	wasmer	iwasm	wasmi	wazero
				Quicksort			
Go	1	0.3333	0.3103	0.3636	0.2857	0.3103	0.6087
wasmtime		1	0.5676	0.7105	0.6	0.6571	0.3902
wasmedge			1	0.5714	0.5862	0.5484	0.4242
wasmer				1	0.6061	0.7188	0.5
iwasm					1	0.7692	0.3636
wasmi						1	0.3429
wazero							1
				Floyd-Warshall			
Go	1	0.3429	0.3214	0.375	0.2963	0.3103	0.5652
wasmtime		1	0.5676	0.7105	0.6	0.6857	0.3902
wasmedge			1	0.5714	0.5862	0.5312	0.4242
wasmer				1	0.6061	0.697	0.5
iwasm					1	0.7407	0.3636
wasmi						1	0.3333
wazero							1
				FASTA			
Go	1	0.3889	0.3333	0.4242	0.3571	0.3226	0.5769
wasmtime		1	0.5676	0.7105	0.6	0.6857	0.3409
wasmedge			1	0.5714	0.5862	0.5312	0.3611
wasmer				1	0.6061	0.697	0.4
iwasm					1	0.7407	0.3056
wasmi						1	0.25
wazero							1
				AES256			
Go	1	0.4	0.3448	0.4375	0.3704	0.3333	0.56
wasmtime		1	0.5676	0.7105	0.6	0.6857	0.3488
wasmedge			1	0.5714	0.5862	0.5312	0.3714
wasmer				1	0.6061	0.697	0.4103
iwasm					1	0.7407	0.3143
wasmi						1	0.2564
wazero							1
				KNN			
Go	1	0.3415	0.303	0.4571	0.2941	0.2778	0.6923
wasmtime		1	0.5385	0.7	0.6053	0.6579	0.3261
wasmedge			1	0.5556	0.6333	0.5	0.4
wasmer				1	0.5405	0.6389	0.425
iwasm					1	0.6774	0.3158
wasmi						1	0.2683
wazero							1
				File System			
Go	1	0.3415	0.303	0.4706	0.2941	0.2778	0.75
wasmtime		1	0.5385	0.675	0.6053	0.7027	0.3409
wasmedge			1	0.5714	0.6333	0.5	0.4242
wasmer				1	0.5556	0.6571	0.4595
iwasm					1	0.6774	0.3333
wasmi						1	0.2821
wazero							1

Table 18. Sørensen-Dice Coefficient. *Generated by the author Safia Guellil.*

	Go	wasmtime	wasmedge	wasmer	iwasm	wasmi	wazero
				Quicksort			
Go	1	0.5	0.4737	0.5333	0.4444	0.4737	0.7568
wasmtime		1	0.7241	0.8308	0.75	0.7931	0.5614
wasmedge			1	0.7273	0.7391	0.7083	0.5957
wasmer				1	0.7547	0.8364	0.6667
iwasm					1	0.8696	0.5333
wasmi						1	0.5106
wazero							1
				Floyd-Warshall			
Go	1	0.5106	0.4865	0.5455	0.4571	0.4737	0.7222
wasmtime		1	0.7241	0.8308	0.75	0.8136	0.5614
wasmedge			1	0.7273	0.7391	0.6939	0.5957
wasmer				1	0.7547	0.8214	0.6667
iwasm					1	0.8511	0.5333
wasmi						1	0.5
wazero							1
				FASTA			
Go	1	0.56	0.5	0.5957	0.5263	0.4878	0.7317
wasmtime		1	0.7241	0.8308	0.75	0.8136	0.5085
wasmedge			1	0.7273	0.7391	0.6939	0.5306
wasmer				1	0.7547	0.8214	0.5714
iwasm					1	0.8511	0.4681
wasmi						1	0.4
wazero							1
				AES256			
Go	1	0.5714	0.5128	0.6087	0.5405	0.5	0.7179
wasmtime		1	0.7241	0.8308	0.75	0.8136	0.5172
wasmedge			1	0.7273	0.7391	0.6939	0.5417
wasmer				1	0.7547	0.8214	0.5818
iwasm					1	0.8511	0.4783
wasmi						1	0.4082
wazero							1
				KNN			
Go	1	0.5091	0.4651	0.6275	0.4545	0.4348	0.8182
wasmtime		1	0.7	0.8235	0.7541	0.7937	0.4918
wasmedge			1	0.7143	0.7755	0.6667	0.5714
wasmer				1	0.7018	0.7797	0.5965
iwasm					1	0.8077	0.48
wasmi						1	0.4231
wazero							1
				File System			
Go	1	0.5091	0.4651	0.64	0.4545	0.4348	0.8571
wasmtime		1	0.7	0.806	0.7541	0.8254	0.5085
wasmedge			1	0.7273	0.7755	0.6667	0.5957
wasmer				1	0.7143	0.7931	0.6296
iwasm					1	0.8077	0.5
wasmi						1	0.44
wazero							1

4.3.7. Conclusions of the performance evaluation

Finally, after analyzing the performance of the .wasm code with the different runtimes, benchmarks, and optimization levels, we present the general conclusions obtained in each section. These conclusions are grouped in relation to execution time, compiled binary size, and system calls.

4.3.7.1. Conclusions: Execution Time

In general, we have observed that it is possible to achieve execution times quite similar to the compiled Go code and, in some cases, even slightly better. The runtimes that have given us the best results have been wasmtime, a project that aims to improve the speed and security of the WASM module, and wazero, a runtime written in Golang.

On the other hand, we observe that the optimizations offered by TinyGo do improve the performance of the code. In fact, in most of the cases, we see that the O0 option, which does not have any optimization, is the one that gives the worst results by far. However, the O2 optimization with gc-leaking, which is recommended by the TinyGo documentation, shows the best results.

Finally, regarding TinyGo's conservative garbage collector, we observed that in general, the options without the GC (gc-leaking) offer better results. Although the impact is greater or lesser depending on the runtime. For example, wasmedge has been the one that has suffered the most from the conservative garbage collector, since it can go from 14 to 3 seconds on average if we switch to gc-leaking. However, for other runtimes such as wazero, wasmtime or wasmer the impact of GC is quite small in a matter of milliseconds.

4.3.7.2. Conclusions: Binary Size

As for the binary size, we observe that despite not applying any optimization (O0), the binary size is reduced by more than 80%. This shows how efficient the TinyGo compiler is, as it is really intended to be run on low-power devices.

In addition, we observe that in general, when applying any optimization (other than O0), we achieve a 10% reduction of the binary size, reaching in some cases more than a 99% total reduction.

On the other hand, we have obtained the best size reduction with OZ optimization and gc-leaking. This result was expected since it is TinyGo's recommendation. Finally, as for the conservative garbage collector, we note that it occupies less than 1% of the code. Therefore, in terms of the size of the binary, it does not affect it significantly.

4.3.7.3. Conclusions: System Calls

Finally, in terms of system calls, we note that wasmedge makes the fewest calls. It is even lower than Golang, and the number of calls is quite stable between the different benchmarks. Therefore, its resource usage is quite low, which makes it less power-consuming and more scalable and portable.

On the other hand, we have the Wazero runtime, which is quite unstable. For example, the machine learning algorithm (KNN) with the optimization O1 reaches 300,000 system calls, a gigantic number. This makes it consume much more OS resources, making it not very suitable for small devices.

As for the optimizations, we have found that, in general, they do not improve or worsen the number of system calls. Although in the case of wazero, as we have mentioned, it is an exception since, sporadically between the different benchmarks, it improves or worsens with one or another optimization.

Finally, thanks to the similarity coefficients, we see that wazero is the most similar to Golang (in terms of the type of system calls) and wasmi the least. This fact could be related to the fact that wazero is written in Go.

4.3.8. Special Benchmarks

After analyzing the performance of the previous benchmarks, we now present two special cases. They are not intended to run them with runtimes and evaluate their performance with the different optimization levels, but to evaluate certain functionalities, which are: parallelism and network communication through sockets.

4.3.8.1. Parallel Matrix Multiplication

This benchmark tries to execute the matrix multiplication algorithm (2000x2000) with parallelism. When running this algorithm in Go, we observe (in Table 19) that there are four CPUs. These correspond to the virtual machine on which we are running these tests. We also observe that as the number of threads increases (variable TH), the variable GOMAXPROCS is modified, "current value" which corresponds to the maximum number of goroutines. At the end, we observe in Figure 45 how the execution time is reduced by increasing threads which is the expected result.

Table 19. Output of the matrix multiplication algorithm with Golang. *Generated by the author Safia Guellil.*

go run benchmark.go 1	go run benchmark.go 2
TH: 1 runtime CPU: 4 GOMAXPROCS set to 1 (current value: 1) Multiplyed 2000x2000 matrices	TH: 2 runtime CPU: 4 GOMAXPROCS set to 2 (current value: 2) Multiplyed 2000x2000 matrices
go run benchmark.go 3	go run benchmark.go 4
TH: 3 runtime CPU: 4 GOMAXPROCS set to 3 (current value: 3) Multiplyed 2000x2000 matrices	TH: 4 runtime CPU: 4 GOMAXPROCS set to 4 (current value: 4) Multiplyed 2000x2000 matrices

However, as for the code compiled with TinyGo, we did not observe the same behavior. We see in Table 20 that despite increasing the number of threads (TH), the variable GOMAXPROCS remains at one, "current value". We can also see in Figure 45 that, despite having four CPUs, the execution time does not decrease as the number of threads increases.

We obtained that result is since TinyGo runs on a single core, in a single thread [45]. Therefore, it does not allow any form of parallelism. However, this is a feature that is on TinyGo's wish list[39]. Therefore, it will probably be implemented in the future.

Table 20. Output of the matrix multiplication algorithm with TinyGo. *Generated by the author Safia Guellil.*

wasmtime ./compiled/benchmark.wasm 1	wasmtime ./compiled/benchmark.wasm 2
TH: 1 runtime CPU: 1 GOMAXPROCS set to 1 (current value: 1) Multiplyed 2000x2000 matrices	TH: 2 runtime CPU: 1 GOMAXPROCS set to 2 (current value: 1) Multiplyed 2000x2000 matrices
wasmtime ./compiled/benchmark.wasm 3	**wasmtime ./compiled/benchmark.wasm 4**
TH: 3 runtime CPU: 1 GOMAXPROCS set to 3 (current value: 1) Multiplyed 2000x2000 matrices	TH: 4 runtime CPU: 1 GOMAXPROCS set to 4 (current value: 1) Multiplyed 2000x2000 matrices

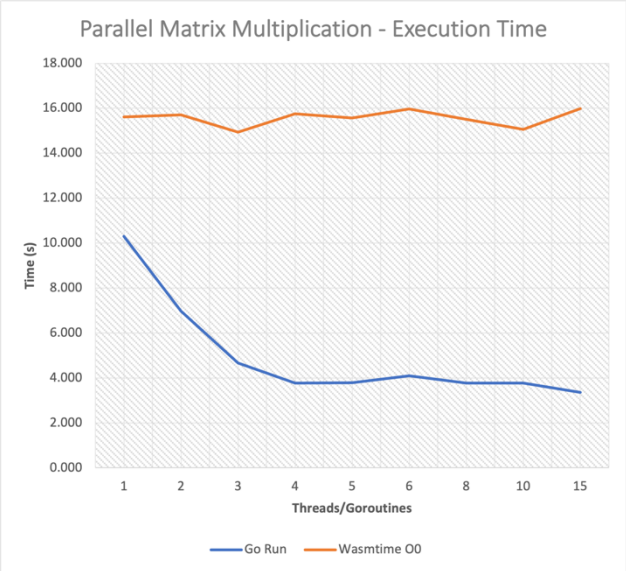

Figure 45. Parallel Matrix Multiplication algorithm comparison with Golang and TinyGo. *Generated by the author Safia Guellil.*

[39] https://github.com/tinygo-org/tinygo/wiki/Wishlist

4.3.8.2. Network Communication (Sockets)

Finally, the second special case that we will analyze is a very simple network communication code that attempts to create a TCP listener on "localhost:3000". Below, you can see the code we used to parse the internally created sockets in the "net.Listener" function. To test it we compile it with TinyGo and run it with the wasmtime runtime.

```
package main
import    "net"

//go:export __test_connection
func connect() {
        conn, err := net.Listen("tcp", "localhost:3000")
        if err != nil {
                println("Error connecting:", err)
                return
        }
        println(conn)
}

func main() {
        connect()
}
```

Output

```
Error connecting: operation not implemented
```

We can observe that when we attempt to run the code, we receive an error from the compiler. This is because the network functionality is not supported yet, since the Web Assembly interface is still in "*preview1*". As you can see in the following code[40] the functions Dial a Listen that accepts or creates networks connections are not implemented and returns an error.

```
func Dial(network, address string) (Conn, error) {
        return nil, ErrNotImplemented
}

func Listen(network, address string) (Listener, error) {
        return nil, ErrNotImplemented
}
```

We can see the corresponding error message on the **tinygo/src/net/errors.go**[41] file, which coincides with the output obtained.

```
package net
import "errors"

var (
        // copied from poll.ErrNetClosing
        errClosed = errors.New("use of closed network connection")

        ErrNotImplemented = errors.New("operation not implemented")
)
```

[40] https://github.com/tinygo-org/tinygo/blob/0d56dee00f49bd50eb373c02c30062a75ec28f10/src/net/dial.go#L16
[41] https://github.com/tinygo-org/tinygo/blob/0d56dee00f49bd50eb373c02c30062a75ec28f10/src/net/errors.go

5. Conclusions

At the beginning of this project, we have established clear and fundamental objectives. First, we have acquired knowledge about Golang programming, the fundamentals of WebAssembly and the basics of the TinyGo compiler. In addition, we have evaluated in depth the automatic memory management of Golang through the garbage collector and how it is inherited to the WASM module. During this analysis we have discovered another interesting functionality "*escape to the heap*" which we have also analysed and evaluated its performance in WebAssembly. After the analysis, we have established different evaluation methodologies and benchmarks to evaluate both the memory management system and the performance of the binaries compiled with TinyGo.

In case of the memory evaluation, we have found that the compiler TinyGo manages the linear memory structure of WebAssembly similarly to what the clang-9 compiler does. Furthermore, through two typical cases, we have found that TinyGo also maintains the "*escape to the heap behavior*" of Golang and even passes it to the WebAssembly module. Finally, as for the garbage collector, we found that since the TinyGo's garbage collector is much simpler, it executes more cleaning cycles and therefore increases the execution time. On the other hand, we could also verify that the functions of this GC are compiled and incorporated into the *.wasm* module, thus providing it with a memory management system.

Regarding the evaluation of performance, we discovered that an execution time similar (or better) to Golang binary can be achieved. This would be with the runtimes: wasmtime a project aimed at improving the speed and security of WebAssembly, and wazero a runtime written in Go. In addition, we found that the levels of optimization offered by TinyGo really improve speed, especially the O2 optimization with gc-leaking recommended by the compiler. On the other hand, we saw that depending on the runtime, the conservative GC of TinyGo could affect the performance significantly; this is the case of the wasmedge runtime. However, in wasmtime, wasmer and wazero the impact is negligible, which provides the WASM module with a memory management system and maintains execution times.

Other aspects evaluated were the size of the binary and the system calls. In terms of size, TinyGo without any optimization, manages to reduce the Golang binary to more than 80% (with optimizations, it can reach more than 99%). This is a clear advantage, as it makes the code more portable. In addition, the optimization with which we obtained the best results was OZ with gc-leaking, which is the one recommended by the compiler. On the other hand, in terms of system calls, we found that the wazero runtime makes the most Go-like calls. However, it is very unstable in terms of the number of system calls and can reach gigantic levels. Another better option is wasmedge, which makes fewer system calls than the source language in Go. This option uses fewer resources and is therefore more suitable for small devices.

Lastly, in relation to the special cases, we note that TinyGo does not allow parallelism since it runs on a single CPU and a single thread. On the other hand, we cannot create algorithms with sockets because network communication is not implemented in the standard WASI and therefore, not in TinyGo either.

In summary, TinyGo is a good compiler for WebAssembly as it generates very small binaries, which helps portability. In addition, Golang's memory system and its features are compiled inside the WASM module, thus enriching it with those special functionalities. There is certainly a difference in runtime due to the conservative GC of TinyGo, but it can be compensated by choosing well the optimizations (especially those recommended by the compiler itself) and the runtimes like wasmtime.

In conclusion, we have been able to verify that TinyGo is a good tool to compile to WebAssembly and Golang is a good choice of source language since they enrich the wasm module and allow greater portability and scalability due to the binary size.

5.1. Future Work

As for future work, there are several lines of research. An interesting one would be to compare the TinyGo compiler with the Golang compiler and observe the differences in performance and binary size. We could also run the WASM binaries in different environments and see if the results are similar, or even run them on low-cost devices like the Raspberry Pi and analyze these WASM modules in an IoT setting.

On a personal level, I would like to continue researching this line of WebAssembly. In fact, thanks to this work, I am a participant of the W3C WebAssembly Community Group[42]. This allows me to attend meetings and keep up to date with the latest developments in this field.

[42] https://www.w3.org/community/webassembly/participants

References

[1]. *Ayke van Laethem – Garbage collection in TinyGo*. (2020, September 24). https://aykevl.nl/2020/09/gc-tinygo

[2]. Bit, A. (2020, May 17). *Write Once for Web Assembly, Run On Everything — A Bit*. A Bit. https://baez.link/write-once-for-web-assembly-run-on-everything

[3]. Butcher, M. P. (2022, February 22). *How to Think About WebAssembly (Amid the Hype)*. Fermyon • Experience the Next Wave of Cloud Computing. https://www.fermyon.com/blog/how-to-think-about-wasm

[4]. Bytecodealliance. (n.d.). *wasmtime/docs/WASI-overview.md at main · bytecodealliance/wasmtime*. GitHub. https://github.com/bytecodealliance/wasmtime/blob/main/docs/

[5]. Can, A. (2022, March 12). Memory management in Go - Ali Can - Medium. *Medium*. https://medium.com/@ali.can/memory-optimization-in-go-23a56544ccc0

[6]. Cheney, D. (2013, June 2). *Why is a Goroutine's stack infinite ? | Dave Cheney*. https://dave.cheney.net/2013/06/02/why-is-a-goroutines-stack-infinite

[7]. Clark, L. (2017, 13 junio). *A cartoon intro to WebAssembly - Mozilla Hacks - the web developer blog*. Mozilla Hacks – the Web developer blog. https://hacks.mozilla.org/2017/02/a-cartoon-intro-to-webassembly/

[8]. Clark, L. (2017, March 31). *Where is WebAssembly now and what's next? - Mozilla Hacks - the Web developer blog*. Mozilla Hacks – the Web Developer Blog. https://hacks.mozilla.org/2017/02/where-is-webassembly-now-and-whats-next/

[9]. Clark, L. (2019, 9 agosto). *Standardizing WASI: a system interface to run WebAssembly outside the web - Mozilla Hacks - the Web Developer blog*. Mozilla Hacks – the Web developer blog. https://hacks.mozilla.org/2019/03/standardizing-wasi-a-webassembly-system-interface/

[10]. colaboradores de Wikipedia. (2023). Go (lenguaje de programación). *Wikipedia, La Enciclopedia Libre*. https://es.wikipedia.org/wiki/Go_(lenguaje_de_programaci%C3%B3n)

[11]. *Comparing Go vs. C in embedded applications - Stack Overflow*. (2022, April 4). https://stackoverflow.blog/2022/04/04/comparing-go-vs-c-in-embedded-applications/

[12]. Confreaks. (2014, May 8). *GopherCon 2014 Opening keynote by Rob Pike* [Video]. YouTube. https://www.youtube.com/watch?v=VoS7DsT1rdM

[13]. *Creative DIY Microcontroller Projects with TinyGo and WebAssembly*. (n.d.). Google Books. https://books.google.es/books?id=83QqEAAAQBAJ

[14]. Debnath, M. (2023). Understanding garbage collection in Go. *Developer.com*. https://www.developer.com/languages/garbage-collection-go/

[15]. *Differences from Go*. (2021, September 5). TinyGo. https://tinygo.org/docs/concepts/compiler-internals/differences-from-go/

[16]. Dwen. (2022, February 12). Golang Memory Escape In-Depth Analysis - Dev Genius. *Medium*. https://blog.devgenius.io/in-depth-analysis-of-golang-memory-escape-edfbfb856913

[17]. Eberhardt, C. (n.d.). *What is WebAssembly?* O'Reilly Online Learning. https://www.oreilly.com/library/view/what-is-webassembly/9781492076902/ch04.html

[18]. *Frequently Asked Questions (FAQ) - The Go Programming Language*. (n.d.). https://go.dev/doc/faq

[19]. Gangemi, S. (2022, January 6). An overview of memory management in Go - SafetyCulture Engineering - Medium. *Medium*. https://medium.com/safetycultureengineering/an-overview-of-memory-management-in-go-9a72ec7c76a8

[20]. Gardón, D. S. (2022, 21st March). WebAssembly vs. JavaScript: How do they compare. *Snipcart*. https://snipcart.com/blog/webassembly-vs-javascript

[21]. GeeksforGeeks. (2020). How to return a Pointer from a Function in C. *GeeksforGeeks*. https://www.geeksforgeeks.org/how-to-return-a-pointer-from-a-function-in-c/

[22]. GeeksforGeeks. (2023). Difference between Static Allocation and Heap Allocation. *GeeksforGeeks*. https://www.geeksforgeeks.org/difference-between-static-allocation-and-heap-allocation/

[23]. *Go language features*. (n.d.). TinyGo. https://tinygo.org/docs/reference/lang-support/

[24]. Golang. (n.d.). *all: add GOOS=wasip1 GOARCH=wasm port · Issue #58141 · golang/go*. GitHub. https://github.com/golang/go/issues/58141

[25]. Hawthorne, S. H. (n.d.). A Language Comparison Between Parallel Programming Features of Go and C. *San Jose State University*, *408-924–1000*. https://www.sjsu.edu/people/robert.chun/courses/cs159/s3/S.pdf

[26]. *Heap allocation*. (2021, May 4). TinyGo. https://tinygo.org/docs/concepts/compiler-internals/heap-allocation/

[27]. *How Shopify uses WebAssembly outside of the browser*. (2020, December 18). Shopify. https://shopify.engineering/shopify-webassembly

[28]. *How we're bringing Google Earth to the web*. (2019, June 20). web.dev. https://web.dev/earth-webassembly/

[29]. Kjorveziroski, V., & Filiposka, S. (2023). WebAssembly orchestration in the context of serverless computing. *Journal of Network and Systems Management*, *31*(3). https://doi.org/10.1007/s10922-023-09753-0

[30]. Kumar, K. (2021). Memory allocations in Go. *DEV Community*. https://dev.to/karankumarshreds/memory-allocations-in-go-1bpa

[31]. Lehmann, D. (2020). *Everything old is new again: binary security of {WebAssembly}*. https://www.usenix.org/conference/usenixsecurity20/presentation/lehmann

[32]. *Optimizing binaries*. (2022, September 4). TinyGo. https://tinygo.org/docs/guides/optimizing-binaries/

[33]. Plauska, I., Liutkevičius, A., & Janavičiūtė, A. (2022). Performance evaluation of C/C++, MicroPython, Rust and TinyGo programming languages on ESP32 microcontroller. *Electronics*, *12*(1), 143. https://doi.org/10.3390/electronics12010143

[34]. Ray, P. P. (2023). An overview of WebAssembly for IoT: background, tools, State-of-the-Art, challenges, and future directions. *Future Internet*, *15*(8), 275. https://doi.org/10.3390/fi15080275

[35]. Roberts, S. (n.d.-b). *Shopify: qué es, funcionamiento y ventajas para tu ecommerce*. https://www.cyberclick.es/numerical-blog/shopify-que-es-funcionamiento-y-ventajas-para-tu-ecommerce

[36]. Roy, S. (2022, September 10). [Golang] Garbage Collection in General - Dev Genius. *Medium*. https://blog.devgenius.io/golang-garbage-collection-in-general-c28ae82558c4

[37]. Singh, N. (2022a). WebAssembly for Beginners – Part 2: Goals, key concepts, and use cases. *Geekflare*. https://geekflare.com/webassembly-wasm-goals-key-concepts-use-cases/

[38]. Singh, N. (2022b). WebAssembly for Beginners – Part 1: An Introduction to WASM. *Geekflare*. https://geekflare.com/webassembly-wasm-introduction/

[39]. Stiller, E. (2021, January 21). The "Wasmer" WebAssembly Runtime is Generally Available. *InfoQ*. https://www.infoq.com/news/2021/01/wasmer-generally-available/

[40]. *TinyGo Home*. (n.d.). https://tinygo.org/

[41]. Tinygo-Org. (n.d.-b). *tinygo/src/runtime at release · tinygo-org/tinygo*. GitHub. https://github.com/tinygo-org/tinygo/tree/release/src/runtime

[42]. Tinygo-Org. (n.d.-c). *wasm: switch to wasi_snapshot_preview1 by aykevl · Pull Request #1690 · tinygo-org/tinygo*. GitHub. https://github.com/tinygo-org/tinygo/pull/1690

[43]. Tinygo-Org. (n.d.). *GitHub - tinygo-org/tinygo: Go compiler for small places. Microcontrollers, WebAssembly (WASM/WASI), and command-line tools. Based on LLVM.* GitHub. https://github.com/tinygo-org/tinygo

[44]. *TinyGo*. (s. f.). https://wazero.io/languages/tinygo/#unsupported-system-calls

[45]. *Tips, tricks and gotchas*. (2023, May 19). TinyGo. https://tinygo.org/docs/guides/tips-n-tricks/

[46]. *WebAssembly | MDN*. (2023, May 31). https://developer.mozilla.org/en-US/docs/WebAssembly

[47]. WebAssembly. (2023, February 9). *meetings/wasi/2023/presentations/2023-02-09-gohman-wasi-roadmap.pdf at main · WebAssembly/meetings*. GitHub. https://github.com/WebAssembly/meetings/blob/main/wasi/2023/presentations/2023-02-09-gohman-wasi-roadmap.pdf

[48]. *WebAssembly*. (s. f.). https://webassembly.org/

[49]. *Why Golang is widely used in the DevOps and cloud native space?* (n.d.). thechief.io. https://thechief.io/c/editorial/why-golang-is-widely-used-in-the-devops-and-cloud-native-space/#:~:text=Go%2C%20with%20its%20ability%20to,system%20resources%20to%20run%20it.

[50]. Wikipedia contributors. (2023). Memory management. *Wikipedia*. https://en.wikipedia.org/wiki/Memory_management

[51]. Wikipedia contributors. (2023). WebAssembly. *Wikipedia*. https://en.wikipedia.org/wiki/WebAssembly

[52]. *World Wide Web Consortium (W3C) brings a new language to the web as WebAssembly becomes a W3C recommendation*. (2019, 5th December). W3C. https://www.w3.org/press-releases/2019/wasm/

[53]. Yan, Y., Tu, T., Zhao, L., Zhou, Y., & Wang, W. (2021). Understanding the performance of webassembly applications. *Association for Computing Machinery*. https://doi.org/10.1145/3487552.3487827