PERFORMANCE ANALYSIS OF C VS TINYGO ON THE RP2040 MICROCONTROLLER

by

SAMUEL IVUERAH
URN: REDACTED

A dissertation submitted in partial fulfilment of the
requirements for the award of

BACHELOR OF SCIENCE IN COMPUTER SCIENCE

May 2025

Department of Computer Science
University of Surrey
Guildford GU2 7XH

Supervised by: REDACTED and REDACTED

I declare that this dissertation is my own work and that the work of others is acknowledged and indicated by explicit references.

Samuel Ivuerah
May 2025

Abstract

This project presents a detailed performance comparison between C and TinyGo on the RP2040 microcontroller, evaluating Tinygo's suitability as a practical alternative for embedded systems development. While C remains the standard for low-level, time-critical applications due to its predictable execution and minimal overhead, TinyGo offers benefits in memory safety, binary size, and developer productivity. A suite of software and hardware benchmarks was implemented in both languages, covering core computational tasks (such as sorting algorithms, matrix multiplication, and FFT) and real-world I/O operations (including ADC, GPIO, PWM, UART, I2C, and interrupt handling). Performance metrics—execution time, binary size, RAM usage, and timing variance—were measured under consistent test conditions using physical hardware and disassembly inspection. The results indicate that C outperformed TinyGo in 10 out of 12 benchmarks, with an average execution time advantage of approximately 31%. In real-time tasks such as interrupt handling and PWM setup, C's speed was significantly higher—up to five times faster—highlighting TinyGo's limitations in latency-sensitive scenarios due to its runtime and abstraction overhead. However, TinyGo proved competitive in algorithmic workloads, occasionally matching or exceeding C's performance at higher recursion or iteration depths. Across nearly all benchmarks, TinyGo consistently produced smaller binaries and used less RAM, often with more stable execution times. This analysis shows that while C remains the preferred choice for hard real-time and performance-critical systems, TinyGo offers a compelling alternative for resource-constrained or less timing-sensitive applications. Developers may benefit from hybrid approaches, using each language where its strengths are most applicable. The project provides actionable insights and performance data to inform language selection for embedded systems on modern microcontrollers like the RP2040.

Acknowledgements

I want to thank my supervisors, **REDACTED** and **REDACTED**, for their guidance and feedback throughout this project. Also, I would like to thank the University of Surrey and the Department of Computer Science for providing the hardware resources and facilities. I also want to thank my peers who provided moral support and technical insights. Finally, my final thanks go to my family for their support.

Contents

List of Figures

List of Tables

Glossary

Benchmarking The process of measuring performance metrics under standard conditions.

Microcontroller A compact integrated circuit designed to govern a specific operation in an embedded system.

PWM A method for simulating an analog output using digital means by switching on and off rapidly.

Jitter Variability in timing precision of a signal or process.

Runtime The period during which a program is executing, or the system that manages it (e.g., TinyGo's runtime).

Garbage Collector (GC) A background system that automatically reclaims memory no longer in use. In TinyGo, it introduces unpredictability in real-time tasks.

TinyGo A subset of the Go programming language designed for small embedded devices. Compiles to WebAssembly or native microcontroller binaries.

Memory Model The system that defines how a computer's memory is accessed and managed, including aspects like endianness and data alignment.

Low-Level Optimisation Techniques used in programming to improve the performance and efficiency of software, typically by reducing memory use or processing time.

Peripheral A device or subsystem that connects to and communicates with a microcontroller, such as sensors, actuators, or communication modules.

Concurrency The ability to execute multiple tasks simultaneously, which is a key feature in TinyGo.

Goroutine A lightweight thread used in the Go programming language, including TinyGo, for concurrent function execution.

Latency The time delay between the initiation and the occurrence of a specific event in a system, often critical in real-time systems.

Binary Size The total size of the compiled code, including both the executable instructions and associated data.

Timing Jitter The variability in the time delay between the expected and actual timing of events in a system, critical in real-time applications.

Throughput The rate at which a system or component can process data, often used to measure performance.

Real-Time Operating System (RTOS)
A specialised operating system designed to handle real-time tasks with strict timing constraints.

Abbreviations

ADC	Analog-to-Digital Converter
GPIO	General Purpose Input/Output
UART	Universal Asynchronous Receiver-Transmitter
FFT	Fast Fourier Transform
SDK	Software Development Kit
LSEP	Legal, Social, Ethical and Professional Issues
RP2040	Raspberry Pi's Dual-core Cortex-M0+ MCU
CSV	Comma-Separated Values
UF2	USB Flashing Format (Microsoft)
I2C	Inter-Integrated Circuit (serial communication protocol)
ISR	Interrupt Service Routine
RTOS	Real-Time Operating System
MCU	Microcontroller Unit
DMA	Direct Memory Access
PICO-SDK	Raspberry Pi Pico Software Development Kit
PWM	Pulse Width Modulation
JIT	Just-In-Time
GC	Garbage Collector
DSP	Digital Signal Processing

Chapter 1

Introduction

1.1 Background and Motivation

Embedded systems are the foundational technology behind modern electronics, powering devices from **medical implants** and **autonomous drones** to **industrial automation** and **IoT sensors**. These systems demand precise, low-latency control, efficient resource management, and reliable real-time performance.

Traditionally, **C** has dominated this field due to its direct hardware access, minimal runtime overhead, and predictable memory model, making it ideal for latency-sensitive applications. However, the increasing complexity of embedded software has driven interest in higher-level languages like **TinyGo**, a subset of **Go** adapted for microcontrollers (TinyGo Authors 2024).

TinyGo promises safer memory management and a more modern developer experience, which, in turn, potentially reduces the risk of memory corruption and logic errors that plague solutions using **C**. While it does support goroutines, its concurrency model is cooperative and limited; this study does not explore concurrent workloads, and structured concurrency is not a focus of the evaluation (TinyGo Authors 2025*c*). However, its adoption remains limited in real-time, performance-critical systems, where **C** continues to dominate. This hesitation stems from two key challenges:

- **Ecosystem Maturity and Tooling Gaps**: Unlike **C**, which benefits from decades of optimisation and mature compiler support, **TinyGo** is a relatively young, community-driven project. Its toolchain, while steadily improving, still lacks the mature, fine-grained control

over peripherals and extensive low-level optimisations found in established C toolchains, potentially limiting its suitability for applications requiring precise timing and predictable real-time performance.

- **Real-Time Determinism**: TinyGo's runtime introduces non-deterministic behaviour through features like garbage collection (GC) and cooperative multitasking via goroutines. While improving developer productivity, these abstractions can lead to unpredictable execution times, potentially violating the strict timing guarantees required in real-time control systems. For example, in applications like **autonomous vehicles**, **medical devices**, and **robotics**, even slight variations in interrupt latency or task execution can disrupt control loops, degrade signal quality, or cause system instability. Moreover, slight variations are particularly critical in safety systems, such as brake-by-wire in modern cars, where response times as low as 150 ms are required to ensure safe operation (AG 2016).

The **RP2040** microcontroller, developed by Raspberry Pi Ltd. (Raspberry Pi Ltd. 2021), presents an ideal platform for this evaluation. It features dual Cortex-M0+ cores, flexible peripherals, and a robust developer community, supporting both **C** (via the Pico SDK (Raspberry Pi Ltd. 2023)) and **TinyGo**. Given its widespread use in prototyping and production, understanding the performance trade-offs between these languages on the RP2040 is highly relevant for both hobbyists and professional developers. Making an informed choice between **C** and **TinyGo** is not just a question of development speed or code safety but also of operational reliability in real-world deployments, where software failures can have severe consequences.

Broader Implications: The implications of this comparison extend beyond academic interest. As microcontrollers increasingly support wearable health monitors, smart city infrastructure, and distributed edge nodes, developers are under pressure to choose tools that balance performance, safety, and ease of use. If languages like **TinyGo** can deliver near-C performance while reducing complexity and improving memory safety, this could significantly improve firmware maintainability and resilience in long-lived, safety-critical deployments (Mastery 2023).

1.2 Problem Statement

While several studies have examined languages like **Rust**, **Python**, and **C++** for embedded systems, there remains a significant gap in benchmark-driven evaluations comparing **TinyGo** and

C on mainstream microcontrollers like the RP2040. For instance, Plauska et al. benchmarked C/C++, MicroPython, Rust, and TinyGo on the ESP32, focusing on signal processing tasks, but omitted peripheral benchmarks such as GPIO and UART, which are critical for real-world applications (Plauska, Liutkevičius & Janavičiūtė 2023): the lack of precise, task-based performance data forces developers to rely on anecdotal evidence to make informed language choices for performance-critical applications. The project aims to fill this gap by providing detailed insights for gathered metrics such as execution time, memory usage, and peripheral performance across a diverse set of tasks; each selected to capture a critical aspect of microcontroller performance:

- **Core Software Tasks**: Recursive algorithms like Fibonacci capture function call overhead and stack management, while matrix multiplication tests data locality and nested loop efficiency, essential for real-time digital signal processing (DSP) and control systems.
- **Peripheral Benchmarks**: GPIO toggling assesses direct digital I/O control latency, PWM tests the responsiveness of hardware-timed waveforms, and interrupts capture real-time event handling efficiency. These benchmarks provide a direct measure of the microcontroller's capability to handle external events with precise timing, which is critical for applications like motor control, pulse-width modulated audio, and high-speed communication.

Unique Contributions

- **Disassembly Analysis**: The project examines generated machine code to provide context for performance variations, thereby revealing the impact of language abstractions on execution speed and memory access patterns.
- **Custom Hardware Instrumentation**: Utilises a purpose-built GPIO pico probe for precise PWM verification, capturing microsecond-level timing data often missed by software profilers.
- **Mixed Benchmarking**: Covers both core algorithms (e.g., FFT, matrix multiplication) and real-time peripheral interactions (e.g., GPIO, UART, interrupts), providing a more holistic view of language-level trade-offs in embedded systems.

1.3 Aim and Objectives

The primary aim of this project is to assess the viability of **TinyGo** as a practical alternative to **C** for embedded system development on the RP2040 microcontroller, and this will be achieved through the following objectives:

- Conduct precise, repeatable benchmarks for core software tasks (e.g., sorting, matrix multiplication, FFT) and peripheral interactions (e.g., GPIO, PWM, UART, I2C).
- Quantify binary size, RAM usage, and runtime jitter to evaluate resource efficiency.
- Identify practical challenges and limitations when implementing and benchmarking.
- Use disassembly insights to provide context for observed performance differences.
- Assess language suitability for real-time constraints, including interrupt handling and low-latency peripheral access.
- Provide actionable recommendations for developers considering TinyGo for performance-critical applications.

1.4 Scope and Limitations

The study focuses on the Raspberry Pi Pico microcontroller, evaluating performance across various software and hardware benchmarks, including loop overhead, recursion, matrix multiplication, FFT, GPIO, UART, and I2C, between **C** and **TinyGo**.

However, several constraints were considered:

- **GPIO:** PWM timings were effectively captured using the GPIO pico probe, which reliably detected rising and falling edges but was unsuitable for GPIO toggle speed measurements, as it could not keep up with the RP2040's maximum toggle rate.
- **Peripheral Performance (PWM Frequency Control)**: TinyGo abstracts peripheral control more heavily than C, particularly for PWM setup, where its machine API lacks direct access to hardware clock dividers, limiting the ability to tune PWM frequencies precisely and, thus, impacting the precision of waveform generation, which is critical for applications like motor control, audio synthesis, and signal modulation.

- **Disassembly Analysis:** While disassembly files were not used for precise cycle counting, it helped provide context for observed performance differences and, thus, highlighted the impact of higher-level language abstractions like garbage collection and type safety.
- **Toolchain Maturity:** Although generally stable, **TinyGo** remains a community-driven project and lacks some fine-tuned control and mature optimisation in traditional **C** toolchains, requiring careful consideration when implementing low-level drivers.

Risks and Challenges

Several challenges were encountered, including:

- **Real-Time Determinism and Garbage Collection Overhead**: TinyGo's garbage collector, though minimal, can still introduce unpredictable latency, violating the hard real-time constraints required for control loops or high-speed data acquisition. Its cooperative multitasking model further complicates timing precision, as context switches are less predictable than traditional interrupt-driven approaches.
- **Debugging Complexity:** Interpreting disassembled TinyGo binaries proved challenging, as the higher-level abstractions obscure precise control flow, complicating performance analysis.
- **Instrumentation Limitations:** The GPIO probe Pico, initially intended for the GPIO toggling benchmark as a substitute for an oscilloscope, proved inadequate for capturing the RRP2040's maximum toggle rates. However, it was repurposed for PWM verification, where the slower, more predictable nature of the signal allowed for more accurate measurement. GPIO code was further validated by intentionally reducing toggle frequencies, providing partial confidence in timing behaviour despite the lack of higher precision test equipment.

1.5 Structure of the Dissertation

This dissertation is structured as follows:

- **Chapter 2** reviews existing literature on embedded languages, performance benchmarking, and prior microcontroller studies.

- **Chapter 3** outlines the methodology, including benchmark design, software and hardware setups, and the measurement techniques used.
- **Chapter 4** presents the graphed results, including execution times, memory usage, and binary sizes.
- **Chapter 5** provides an analysis and critical evaluation between **C** and **TinyGo**, and the project as a whole.
- **Chapter 6** discusses the project's legal, social, ethical, and professional implications, including potential impacts on sustainability.

Study Rationale and Context

This project is situated within the broader mobile and embedded computing field, where power efficiency, size constraints, and real-time responsiveness are paramount. It also aligns with the emerging edge computing trend, which pushes intelligence closer to the data source—often onto microcontrollers like the RP2040. Evaluating modern language toolchains like **TinyGo** in comparison with **C** contributes to ongoing debates about language evolution in constrained environments and intersects with research in IoT systems, energy-aware programming, and runtime safety in mobile computing (Mastery 2023).

As outlined above, the project aims to address the critical gap in real-world, hardware-focused benchmarking for high-level embedded languages. Previous studies have explored the capabilities of **Rust**, **Python**, and **TinyGo** for embedded applications; however, these studies often focus on software-only metrics, overlooking the critical importance of hardware interactions. The project seeks to bridge this gap by providing a comprehensive evaluation spanning core software algorithms, real-time peripheral control, and disassembly analysis to offer a more complete view of language-level trade-offs for performance-critical applications on mainstream platforms like the RP2040.

Chapter 2

Literature Review

2.1 C in Embedded Systems: Strengths and Risks

The C programming language has long been the de facto standard for embedded systems development: it provides fine-grained control over hardware, deterministic performance, and minimal runtime overhead, essential in real-time and resource-constrained environments. C's low-level memory model and direct register access allow developers to write highly optimised code, making it well-suited for systems with critical latency and power efficiency. However, this flexibility comes at the cost of increased complexity and a higher risk of memory safety issues, as discussed by Microsoft, where approximately 70% of all software security vulnerabilities are rooted in memory safety problems in C and C++ codebases (Cimpanu 2019).

In contrast, **TinyGo** represents a more modern approach to embedded programming, leveraging Go's memory safety, static typing, and garbage collection to reduce common programming errors. Go's design prioritises simplicity, safety, and concurrency, with a memory model that abstracts away many of the pitfalls of manual memory management while still providing the efficiency required for microcontroller applications (TinyGo Authors 2024). However, this abstraction can obscure low-level hardware interactions critical in real-time systems. As noted by Elliott et al., embedded languages often struggle to balance high-level expressiveness with low-level efficiency, as the abstractions that simplify code can also introduce significant overhead, impacting execution speed and memory usage (Elliott, Finne & de Moor 2003).

2.2 TinyGo as a Modern Embedded Language

TinyGo is a modern alternative for embedded development, offering a subset of the Go programming language compiled into lightweight binaries suitable for microcontrollers (TinyGo Authors 2024). TinyGo aims to combine Go's developer-friendly syntax and concurrency features with the low-level control required in embedded systems: its memory safety, static typing, and structured concurrency provide a higher-level development experience, while its compact runtime allows it to run on microcontrollers with minimal RAM and flash requirements.

However, these benefits come with trade-offs that directly impact real-time performance. For instance, TinyGo's garbage collector, albeit minimal compared to Go's GC, can still introduce unpredictable pauses, complicating the precise timing required for control loops or signal processing. Additionally, its machine API abstracts peripheral interactions, which can potentially add latency and, in turn, reduce the predictability critical for real-time systems. This trade-off is particularly problematic for applications where timing jitter and response consistency are essential, such as motor control, robotics, or safety-critical medical devices.

Despite this, prior benchmarking studies primarily focused on software-only metrics whilst overlooking these critical real-time constraints. This project directly addresses this gap by evaluating computation speed and low-level peripheral access, interrupt latency, and memory management to provide a more comprehensive view of TinyGo's suitability for embedded systems.

2.3 Benchmarking Studies: C vs TinyGo vs Others

Several benchmarking studies have evaluated programming languages for microcontroller applications. El-Hadedy et al. benchmarked the TinyJAMBU cryptographic primitive across C, Rust, and TinyGo implementations on multiple embedded platforms, including the RP2040. The study concluded that while C was consistently the fastest, TinyGo achieved the lowest power consumption in many cases, highlighting a trade-off between performance and efficiency (El-Hadedy, Hua, Saqib, Yoshii, Hwu & Margala 2023). Their walkie-talkie simulation demonstrated real-time encrypted UART transmission across devices, showcasing TinyGo's potential for low-power secure communication. However, other peripheral tests like ADC or GPIO were not explored.

Plauska et al. (Plauska et al. 2023) evaluated a set of embedded signal-processing benchmarks

(e.g., FFT, CRC-32, SHA-256, IIR/FIR filters) on the ESP32 using C/C++, Rust, MicroPython, and TinyGo. While this provided a broad view of software performance, it omitted key peripheral benchmarks like UART, GPIO, and ADC, which are critical for real-world embedded applications. This omission was partly due to the limited peripheral support available in the TinyGo toolchain at the time and incomplete hardware support on the ESP32 platform itself (Plauska et al. 2023). For example, interfaces like WiFi, Bluetooth, and even ADC were not fully supported, limiting the scope of achievable benchmarks. This is significant, as peripheral latency and I/O speed are often the bottlenecks in performance-critical systems. Without this context, software-only results can be misleading, failing to capture the full impact of higher-level language abstractions on real-time performance. This project addresses this critical gap by including these overlooked benchmarks, providing a more realistic assessment of embedded system performance.

Samefors and Sundman (Samefors & Sundman 2023) studied wake-up response and power consumption on the STM32WB55 using MicroPython and C. Their benchmarks covered interrupt-driven wake-up from LightSleep and DeepSleep modes. Results showed that C had a consistently faster and less jittery response. While the language under test differed, their methodology directly applies to TinyGo's runtime and event handling, highlighting a practical need for benchmarking latency and responsiveness in interpreted or semi-compiled languages.

An earlier review by Powers (Powers 1998) underscores the historical difficulty of benchmarking embedded systems using general-purpose metrics. While widely used, he argued that desktop-centric benchmarks such as Dhrystone or SPECint fail to capture the I/O latency, peripheral integration, and interrupt responsiveness that dominates real-world embedded workloads. His review advocated for **system-level benchmarks** that combine CPU performance with peripheral response metrics and application-specific behaviours. This aligns with the motivation of this project, which aims to go beyond algorithmic benchmarks by incorporating hardware-interfacing tasks such as ADC sampling, GPIO toggling, PWM configuration, and interrupt latency. Powers' work reinforces that such holistic benchmarking approaches remain essential — and underexplored — even decades later.

Table 2.1: Comparison of Benchmark Studies on Embedded Languages

Study	**Benchmarks Conducted**	**Languages**	**Platform**	**Limitations**
Plauska et al. (2023)	CRC, SHA-256, FFT, FIR/IIR	C, Rust, Python, TinyGo	ESP32	✗ UART, ADC, GPIO not tested
El-Hadedy et al. (2023)	TinyJAMBU cipher, UART	C, Rust, TinyGo	RP2040	✗ No GPIO, ADC, PWM
Samefors and Sundman (2023)	Sleep/wake latency, power	C, MicroPython	STM32WB55	✗ No crypto or comm. tasks
This Study	Full I/O stack (GPIO, PWM, ADC, UART, I2C, Interrupts)	**C, TinyGo**	**RP2040**	✓ All tested

2.4 Ecosystem Maturity and Tooling

C has long been the dominant language in embedded systems, benefiting from a highly mature ecosystem with robust compiler support, extensive debugging tools, and comprehensive peripheral libraries. For instance, the Raspberry Pi Pico SDK offers direct control over RP2040 hardware via a comprehensive API (Raspberry Pi Ltd. 2023). This level of control is critical in applications where even microsecond-level delays can compromise system safety, as seen in brake-by-wire systems and medical devices, where strict timing guarantees are non-negotiable (AG 2016).

TinyGo, by contrast, offers a more modern, high-level approach but lacks the same depth of hardware control. However, its ecosystem is rapidly improving, with ongoing efforts to optimise code size, reduce power consumption, and address common programming caveats (TinyGo Authors 2025*b*, TinyGo Authors 2025*a*, TinyGo Authors 2025*c*). Despite these improvements, the language still abstracts peripheral interactions through its machine API, potentially adding latency and reducing the predictability required for real-time systems (TinyGo Authors 2024).

The project aims to address this gap by directly evaluating the impact of these abstractions on real-world performance, including peripheral latency, interrupt handling, and memory management.

Table 2.2: Feature Comparison: C vs TinyGo for Embedded Development

Feature	**C**	**TinyGo**
Memory Management	Manual	Garbage-collected (non-deterministic)
Peripheral Access	Full register control	Abstracted via `machine` API
Toolchain Maturity	Very mature	Actively evolving
Real-Time Capabilities	High (predictable)	Limited (GC pauses)
Concurrency Model	Manual (ISRs, threads)	Cooperative goroutines

2.5 Safety, Predictability and Runtime Guarantees

TinyGo provides memory safety through Go's static typing and avoids common pitfalls in C, such as null pointer dereferencing and buffer overflows. However, this safety comes at the cost of non-deterministic garbage collection, which can introduce unpredictable pauses. For example, brief interruptions can disrupt control loops, cause missed sensor inputs, or degrade signal quality in latency-sensitive applications like motor control, robotics, or industrial automation. This project will directly measure these impacts through carefully designed latency benchmarks to quantify the real-world trade-offs. This is a known limitation for languages with automatic memory management, as any non-deterministic pause can lead to missed deadlines or jitter in time-critical applications. This challenge is compounded when embedded languages rely on garbage collection, as highlighted by Elliott et al., who noted that even with aggressive compiler optimisations, the overhead of memory management can still significantly impact real-time performance (Elliott et al. 2003). Similarly, the Delite framework, designed for high-performance DSLs, also struggled with latency control, despite sophisticated code generation techniques (Sujeeth, Lee, Brown, Rompf, Chafi, Wu, Nowak, Odersky & Olukotun 2014).

Community insights from experienced TinyGo developers echo these concerns, noting that while TinyGo is capable of soft real-time performance, its garbage collector is not optimised for the low-latency, deterministic behaviour required for hard real-time systems. For example, one developer reported a consistent 5 microseconds scheduling overhead per goroutine context switch on the

RP2040, highlighting a practical limitation for timing-critical tasks (Reddit Users 2023).

By contrast, C allows complete control over memory and execution, supporting microsecond-level deterministic scheduling. While this increases complexity and risk of errors, it provides the predictability required by safety-critical systems. Historical examples like TinyOS exemplify trade-offs in embedded memory safety and minimal abstraction (Gay, Levis & Culler 2006).

2.6 Summary

In this review, we have highlighted the key trade-offs between C and TinyGo regarding embedded systems development. Whilst C continues to be the standard for hard and soft real-time performance, along with its precise timing and hardware control, other solutions, such as TinyGo, can offer significant advantages in developer productivity, memory safety, and concurrency. However, these benefits may come with trade-offs that must be carefully considered in real-time performance. The project addresses these gaps by providing a more comprehensive evaluation of both languages, including their impact on peripheral latency, interrupt handling, and memory management. Although Plauska et al. (Plauska et al. 2023) and El-Hadedy et al. (El-Hadedy et al. 2023) contribute valuable insights, both focus primarily on software or crypto benchmarks, omitting many hardware-level tasks. The study directly addresses these limitations by providing broader benchmarks (such as software and hardware tasks covering GPIO, PWM, ADC, UART, I2C, and external interrupt handling). Moreover, the work includes a detailed analysis of disassembled machine code to contextualise observed performance differences, offering a more comprehensive evaluation of embedded language suitability for real-time applications. Powers (Powers 1998) noted that CPU-focused benchmarks often overlooked embedded concerns like ISR latency and peripheral coordination. His call for realistic, system-level benchmarks has seen limited uptake, making it timely to revisit these principles with modern microcontrollers and languages. This project answers that call by evaluating trade-offs between C and TinyGo, incorporating peripheral interaction alongside control flow metrics. This approach builds on the insights of Elliott et al., who demonstrated that higher-level abstractions can introduce significant performance costs if not carefully managed during compilation (Elliott et al. 2003). Similarly, the Delite framework's challenges with efficiently compiling domain-specific languages underscore the need for careful performance tuning, even in environments where code generation is heavily optimised (Sujeeth et al. 2014).

Chapter 3

Methodology and Implementation

3.1 Introduction

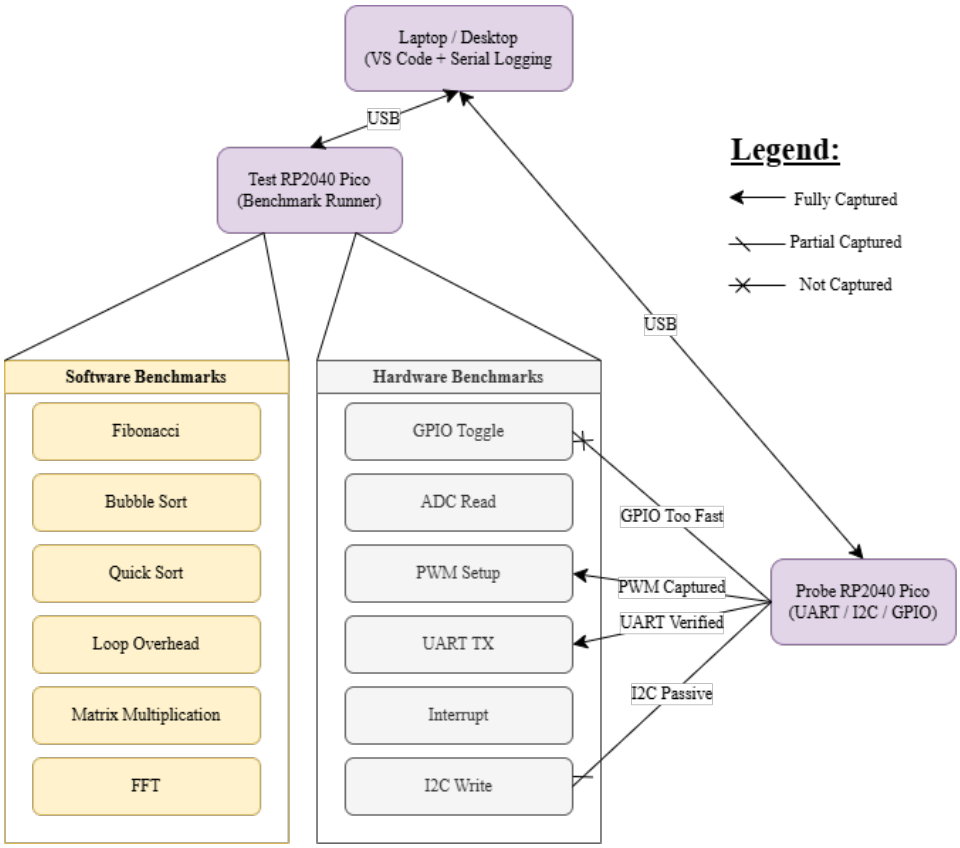
This chapter presents the methodology and implementation strategy used to benchmark **C** and **TinyGo** on the **RP2040** microcontroller. The aim is to provide a comprehensive, reproducible evaluation of both languages, covering execution time, memory usage, and peripheral performance across real-world embedded tasks. This includes both **software** benchmarks (e.g., sorting algorithms, matrix multiplication, FFT) and **hardware** benchmarks (e.g., GPIO toggling, UART, I2C), reflecting the diverse requirements of performance-critical embedded systems.

To achieve this, both software routines and hardware-interfacing tasks were implemented to capture computational performance and how each language manages peripheral control in a real embedded setting. The approach outlined throughout this chapter enables us to extract detailed insights into binary size, RAM usage, runtime jitter, and low-level timing behaviour whilst supporting the aim of providing precise, quantitative comparisons between C and TinyGo.

3.2 System Overview

The benchmarking system is based on a dual-Pico architecture, where a dedicated **Test RP2040 Pico** executes all benchmarks, whilst a secondary **Probe RP2040 Pico** provides supplemental signal observation, as illustrated in Figure 3.1. This design allows the Test Pico to capture precise timing data and output over USB. At the same time, the Probe Pico provides additional insights into peripheral behaviour without interfering with primary benchmark execution.

Figure 3.1: High-level experimental setup for C and TinyGo benchmarking on the RP2040.



3.2.1 Test Pico Design

The **Test Pico** handles all benchmark execution, including software and hardware tasks. It provides timestamped performance data over USB serial, capturing key metrics like execution time, memory usage, and code size. The Test Pico is accountable for all primary measurements and is the site of all benchmarking tasks.

3.2.2 Probe Pico Design

The **Probe Pico** acts as a passive observer, providing supplemental insights into peripheral behaviour. It is used in three primary roles:

- **PWM Verification:** Provides a crude view of the PWM signal shape, capturing basic waveform characteristics.
- **UART Validation:** Verifies UART message transmission without introducing significant latency.

- **I2C Acknowledgement:** Acts as a passive I2C device, acknowledging incoming transmissions to confirm correct bus operation. It does not respond with data but is a lightweight signal presence check.

The Probe Pico is purely observational, ensuring it does not interfere with the primary performance metrics captured by the Test Pico. It is not used for direct timing analysis but provides valuable context for verifying peripheral behaviour.

3.2.3 Measurement Context

While the Test Pico captures all primary performance data, the Probe Pico provides supplemental signal validation where external observation is required. For example, it can confirm UART message integrity, observe basic PWM signal structure, and passively acknowledge I2C transmissions. However, it does not perform direct timing measurements, which would conflict with the primary focus on execution time and memory profiling.

3.2.4 Development Environment

The benchmarks were developed using a combination of the official Pico SDK for C and the TinyGo compiler, each configured for size-optimised builds. Development and debugging were performed using VS Code, with USB serial output captured for analysis. As detailed in later sections, the Probe Pico firmware was developed separately to support multiple operational modes (e.g., UART logger, I2C responder, PWM signal checker).

This dual-Pico architecture, combined with structured logging and isolated signal observation, provides a robust foundation for evaluating language-level trade-offs in real-world embedded contexts.

3.3 Requirements and Design Choices

This section outlines the primary project goals and the key design choices made to achieve accurate, repeatable benchmarks for evaluating the performance of **C** and **TinyGo** on the RP2040 microcontroller.

3.3.1 Project Goals

- **Comprehensive Benchmark Coverage:** Conduct precise, repeatable benchmarks for core software tasks (e.g., sorting, matrix multiplication, FFT) and peripheral interactions (e.g., GPIO, PWM, UART, I2C) to capture the full spectrum of embedded workloads.
- **Resource Efficiency Assessment:** Quantify binary size, RAM usage, and runtime jitter to evaluate the resource efficiency of each language.
- **Practical Constraint Identification:** Identify practical challenges and limitations encountered during real-world testing, including hardware-specific constraints and language-level overheads.
- **Disassembly Analysis:** Make use of disassembly insights to contextualise observed performance differences.
- **Real-Time Suitability Testing:** Assess the suitability of each language for real-time constraints, including interrupt handling and low-latency peripheral access.
- **Developer Guidance:** Provide actionable recommendations for developers considering TinyGo for performance-critical applications based on quantitative benchmark data.

3.3.2 Design Choices

- **Dual-Pico Architecture:** Chosen to minimise interference during timing measurements, with a dedicated **Test Pico** for benchmarks and a passive **Probe Pico** for signal verification.
- **LLVM-Based Memory Profiling:** `llvm-size`, `llvm-nm`, and `llvm-objdump` were used for code size analysis, providing precise function-level memory breakdowns.
- **Standardised CSV Logging:** All benchmarks output results in a consistent CSV format, simplifying post-processing and reducing the risk of data misinterpretation.
- **Modular Build Systems:** Separate build configurations for C and TinyGo benchmarks were used to prevent cross-contamination and ensure accurate memory measurement.

  While each component used in the benchmarking process (e.g., `llvm-objdump`, standard timers, dual microcontrollers) is individually standard, their deliberate integration into a self-contained, interference-minimised benchmarking framework tailored for the RP2040 microcontroller represents a novel combination. This approach enables high-resolution,

language-level performance comparisons between **C** and **TinyGo**, while minimising the risk of measurement contamination and allowing reproducible evaluation across multiple benchmark categories.

3.4 Methodology

3.4.1 Compiler Optimization

Both the C and TinyGo benchmarks were compiled using the default optimisation settings to reflect realistic embedded use cases:

- **C**: Compiled with the Pico SDK's default `-O2` flag, which balances execution speed and code size without aggressive inlining. This is a typical setting for embedded projects, where performance and memory efficiency are critical.
- **TinyGo**: Compiled with the default `-opt=z` flag, which prioritises binary size reduction by limiting inlining and loop unrolling. This setting is the closest match to the default C optimisation level, ensuring a fair comparison.

These default optimisation levels were chosen to reflect typical embedded usage, where code size and execution speed must be carefully balanced against available memory and processing power.

3.4.2 Repository Structure

The project's codebase is organised into three repositories, each structured for isolation, reproducibility, and modular benchmarking:

- `c_benchmarks/` — C implementations for all software and hardware benchmarks.
- `tinygo_benchmarks/` — Equivalent TinyGo implementations, designed with matching benchmark structure.
- `tools/` — Auxiliary programs for peripheral signal logging (e.g., UART logger, I2C responder, GPIO probe).

Each repository adheres to a consistent internal layout:

```
<repo>/
  src/              % One folder per benchmark
    adc/
    gpio/
    ...
  results/          % CSV logs and memory data
    raw/
    logger/
    memory/
  build/            % Compiled binaries (TinyGo only)
  include/          % (C only) Shared headers
  CMakeLists.txt    % (C only) Build config for Pico SDK
  tools.c           % (tools repo only) Entry point with mode switch
```

All benchmarks are compiled as standalone binaries to isolate results and prevent cross-contamination. Detailed build and flashing instructions are provided in each repository's `README.md` to ensure full reproducibility.

3.4.3 Build and Flashing Workflow

Each repository uses a dedicated build system optimised for embedded benchmarking and reproducibility. All benchmarks were compiled as standalone binaries to eliminate cross-contamination of unused code and simplify disassembly and memory measurement.

C Benchmarks

C benchmarks were compiled using the official Raspberry Pi Pico SDK and `CMake`. Builds were configured manually in `CMakeLists.txt` to include only the relevant benchmark source files, ensuring accurate binary size measurement. The project used the official VS Code Raspberry Pi Pico extension to streamline building and flashing during development (Raspberry Pi Foundation 2023).

TinyGo Benchmarks

TinyGo benchmarks were compiled using the `tinygo build` command targeting the RP2040 microcontroller via `-target=pico`. A Windows batch script `build.bat` was created to simplify benchmark selection and compilation for rapid iteration.

Tool Firmware

The `tools/` repository contains peripheral utilities such as a GPIO edge logger, UART logger, and passive I2C responder. Each is located in its own folder, and the active utility is selected

via a macro in `tools.c`. The tools were compiled using the same Pico SDK configuration as the C benchmarks, ensuring consistent hardware-level timing behaviour.

3.4.4 Memory Analysis

Binary memory usage and layout were analysed using the following LLVM tools:

- `llvm-size` – Reports overall section sizes for each compiled binary.
- `llvm-objdump` – Used for disassembly and function-level breakdown.
- `llvm-nm` – Lists static symbols and their sizes.

These tools enabled detailed inspection of code size, stack depth, and static memory allocations, and were applied consistently to both C and TinyGo binaries.

3.4.5 Data Logging

Each benchmark prints its results over USB serial in a consistent, structured CSV format. Serial output was viewed directly using VS Code's built-in serial monitor. This approach—built on hardware-backed timing functions described in Section 3.4.5.1—ensures comparability across languages and benchmark types. The standard format simplifies downstream parsing and plotting:

```
task,method,size,time_us
bubblesort,bubble,50,4801
fibonacci,recursive,20,6765,2660
adc,single_read,1000,11
```

This structure was chosen to make analysis and visualisation straightforward. Memory profiling results were separately captured using `llvm-size`, `llvm-objdump`, and `llvm-nm`, providing detailed insight into code structure and compiled output size.

3.4.5.1 Timing Consistency

To ensure a fair comparison between C and TinyGo implementations, all benchmarks were timed using native microsecond-resolution functions specific to the RP2040 platform.

- In **C**, all measurements used `get_absolute_time()` and `absolute_time_diff_us()` from the Pico SDK, or alternatively `time_us_32()` where appropriate for single-point reads.

- In **TinyGo**, all timing was handled using `time.Now()` and `time.Since()` with final conversion to microseconds via `.Microseconds()`.

Both approaches use hardware-backed timers and avoid coarse system delays, ensuring accurate, low-jitter timing suitable for microsecond-scale benchmarking across all tests.

3.4.6 Testing Procedure and Consistency

To ensure reliable, repeatable results across benchmarks, a standardised testing procedure was adopted:

- **Benchmark Repetition:** Each software and hardware benchmark was executed **5 times**, with a **1-minute pause** between runs to avoid thermal effects, state contamination, and USB serial buffering issues. This interval ensures that each run is effectively isolated, reducing the risk of performance interference from residual heat or cached state.
- **Interrupt Latency Testing:** For the **Interrupt Latency** benchmark, the test button was manually pressed **5 times**, with each press triggering an interrupt on **GPIO14** (Pin 19) and producing an audible buzzer signal on **GPIO15** (Pin 20). To avoid overlapping interrupts, the next button press was delayed until a USB message confirmed the completion of the previous ISR, ensuring accurate timing capture without race conditions.
- **USB Serial Stability:** All benchmarks were configured to output results via USB serial, with each run generating CSV data. The approach prevents partial transmission errors and aligns with the project's focus on reproducible, isolated measurements.
- **Data Consistency:** To further enhance consistency, benchmarks were executed in the same physical environment, with no additional loads on the host PC, and the Pico was disconnected from other peripherals during active measurement phases. This reduces the risk of electrical interference and ensures a stable power supply, critical for accurate timing.

3.4.7 Validation and Testing Coverage

While this project focuses on performance benchmarking rather than functional development, several validation strategies were implemented to ensure accuracy and repeatability of results.

3.4.7.1 Correctness Validation (Functional Testing)

- **Software Benchmarks**: For algorithms such as Fibonacci, Quick Sort, and Matrix Multiplication, the correctness of output was verified against known expected values. For example, Fibonacci outputs for $n = \{10, 20, 30, 35\}$ were verified in both C and TinyGo implementations. Each output was printed alongside execution time to demonstrate agreement in results across languages, e.g., `fib(35) = 9227465` in both cases.
- **UART Transmission**: A dedicated UART logger running on a second Pico confirmed the transmission of 100 messages without corruption. The logger recorded all received bytes and ensured full payload integrity.
- **PWM Waveform Validation**: The Probe Pico was used to verify correct PWM waveform generation. Although unable to capture high-frequency transitions, it successfully confirmed period and duty cycle alignment at a reduced frequency setting.

3.4.7.2 Timing Validation

- All timing measurements were cross-validated by capturing multiple samples (five repetitions per benchmark) and comparing USB-logged durations with expected control flow complexity. For example, recursive Fibonacci execution time increased exponentially with n, while iterative timings remained near constant.

3.4.7.3 Test Repeatability and Isolation

- Benchmarks were executed in five independent runs with cold starts and one-minute gaps to eliminate thermal effects, cache interference, or USB buffering anomalies.
- Each benchmark was compiled in isolation (standalone UF2), ensuring no residual code or state contaminated the measurement.

3.4.8 Challenging Aspects

While the benchmarking system was designed for repeatability and minimal interference, several practical and technical challenges were encountered:

- **TinyGo PWM Limitations:** The `machine.PWM` interface in TinyGo does not expose

low-level configurations such as clock divider settings, thus making it difficult to precisely match the output frequency generated in the C implementation. This discrepancy is further discussed in Section 5.4.3.

- **Manual Flashing:** The lack of automation in the RP2040's flashing process, which requires entering BOOTSEL mode manually, introduced delays in testing and prevented full scripting of benchmark cycles.
- **Peripheral Verification Without Intrusion:** Capturing signal-level behaviours without introducing performance artefacts required careful isolation using a second Pico. Even minor changes in electrical load during debugging had to be accounted for and mitigated.
- **Interrupt Timing Precision:** Measuring ISR latency in a low-jitter, repeatable way demanded precise control of test conditions, including button debounce and accurate timestamping before polling.
- **GPIO Probe Limitations:** The Probe Pico was initially used to validate GPIO toggle behaviour but could not reliably capture the RP2040's maximum toggle rates. It was later repurposed for slower PWM signal checks, where the reduced frequency allowed more meaningful observation. Reduced-frequency test runs provided partial validation in lieu of high-precision instrumentation.

These limitations were either mitigated or explicitly accounted for in the design and analysis phases to maintain the reliability and validity of all collected performance data.

3.5 Software Benchmarks

3.5.1 Unified Purpose and Rationale

The software benchmarks chosen for this project represent diverse computational patterns in embedded systems: these tasks capture the core challenges of microcontroller programming, including recursion, memory access, nested loops, and signal processing. Each benchmark tests different aspects of the RP2040's performance, providing a comprehensive view of language-level trade-offs between C and TinyGo.

3.5.2 Fibonacci

3.5.2.1 Purpose

The Fibonacci sequence is a canonical example of a mathematical recurrence relation, defined as:

$$F(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F(n-1) + F(n-2) & \text{otherwise} \end{cases}$$

This benchmark captures the cost of function calls and stack management, as it can be implemented recursively and iteratively. The recursive form is beneficial for testing function call overhead, stack depth, and branching efficiency, while the iterative form serves as a baseline for comparison. C and TinyGo tested this for inputs $n = 10, 20, 30, 35$, allowing direct comparison across increasing computational loads.

3.5.2.2 Critical Differences

In C, the recursive and iterative versions were implemented as separate functions: `fib_recursive(int n)` and `fib_iterative(int n)`. The recursive version directly mirrors the mathematical definition by using nested function calls and simple branching. Whereas, the iterative version uses a single loop and two integer accumulators for more efficient execution.

In TinyGo, these functions were implemented as `FibRecursive(n int)` and `FibIterative(n int)`. The recursive function is conceptually identical to the C version, relying on Go's built-in function call stack. The iterative version benefits from Go's tuple assignment syntax, allowing more concise variable updates without sacrificing readability. This approach aligns with the typical Go programming style, prioritising readability and simplicity over fine-grained memory control.

3.5.3 Bubble Sort

3.5.3.1 Purpose

Bubble Sort is a foundational sorting algorithm of $O(n^2)$ time complexity. It sorts by comparing adjacent elements in an array and swaps them if they are out of order, effectively "bubbling" the largest elements to the end of the list. This algorithm is computationally expensive for large datasets but is easy to implement, making it a valuable benchmark for evaluating memory access patterns and nested loop performance.

3.5.3.2 Critical Differences

In C, the sorting logic was implemented in `bubble_sort(int* arr, int n)`, using nested `for` loops and a temporary variable for swapping. This approach directly manipulates memory, providing fine control over data movement.

In TinyGo, the equivalent function, `BubbleSort(arr []int)`, used slice syntax and tuple assignment for element swapping. This approach aligns with the Go philosophy of using slices for dynamic memory management and trading raw memory access for safer, more flexible data handling.

3.5.4 Quick Sort

3.5.4.1 Purpose

Quick Sort is a divide-and-conquer sorting algorithm that recursively partitions an array around a pivot element. It typically has an average time complexity of $O(n \log n)$, where it can be reduced to $O(n^2)$ in the worst case if poorly partitioned. This makes it an excellent benchmark for evaluating recursion depth, partitioning efficiency, and memory access patterns.

3.5.4.2 Critical Differences

The C implementation used the functions `quick_sort(int* arr, int low, int high)` and `partition(int* arr, int low, int high)`. The partitioning function follows the Lomuto partition scheme by moving elements around a pivot and recursively sorting the resulting sub-

arrays. This approach relies on direct array indexing for maximum control over memory layout.

In TinyGo, the corresponding functions `QuickSort(arr []int, low, high int)` and `partition(arr []int, low, high int)` use slice syntax, which simplifies indexing but introduces a layer of abstraction over raw memory access. This reflects the typical Go approach, where slices are used for flexible, bounds-checked data structures, reducing the risk of out-of-bounds errors.

3.5.5 Loop Overhead

3.5.5.1 Purpose

This benchmark isolates the control cost of a basic `for` loop by measuring the time required to perform many simple increment operations. It is designed to reveal the raw overhead of loop iteration without the influence of function call overhead or data movement, making it a good baseline for understanding loop control efficiency.

3.5.5.2 Critical Differences

The C version used the function `loop_counter(volatile int* sink, int iterations)`, which explicitly marked the counter as `volatile` to prevent the compiler from optimising away the loop. This approach ensures consistent iteration counts without loop unrolling or dead code elimination.

In TinyGo, the function `LoopN(n int)` relied on a global variable to prevent optimisation, as Go lacks the `volatile` keyword. This approach introduces less direct control over optimisation barriers, reflecting Go's higher-level approach to memory management.

3.5.6 Matrix Multiplication

3.5.6.1 Purpose

Matrix multiplication is fundamental in digital signal processing (DSP), robotics, and scientific computing. It tests the efficiency of nested loops and memory access, as it involves repeatedly accessing and updating elements in large 2D arrays. The operation is defined as:

$$C[i][j] = \sum_{k=0}^{N-1} A[i][k] \cdot B[k][j]$$

This benchmark captures the cost of nested loop execution and data locality, which can significantly impact performance on memory-constrained devices.

3.5.6.2 Critical Differences

The C version used a fixed-size, statically allocated 2D array with direct index-based element access. This approach provides predictable memory access patterns but can result in high memory usage for large matrices.

In TinyGo, the equivalent function, `MatrixMultiply(size int, A, B, C [][]int)`, used slices to represent the matrices. This approach aligns with the Go philosophy of using dynamically sized data structures, allowing more flexible memory management at the cost of some additional allocation overhead.

3.5.7 FFT (Radix-2)

3.5.7.1 Purpose

The Fast Fourier Transform (FFT) is a critical algorithm in digital signal processing (DSP) that extracts frequency components from a time-domain waveform. This benchmark implements the radix-2 Cooley-Tukey FFT algorithm, which operates in $O(n \log n)$ time and is efficient for input sizes that are powers of two.

3.5.7.2 Critical Differences

The C implementation used the function `fft_radix2()`, which relies on fixed-size arrays and direct memory access for efficient in-place computation. It also includes a bit-reversal step for input reordering, a key step in radix-2 FFT algorithms.

In TinyGo, the equivalent function, `fftRadix2()`, used slices for dynamic memory allocation and relied on the `float32` type for all internal calculations. This approach aligns with Go's slice-based data structures, prioritising flexibility and safety over raw memory access.

3.6 Hardware Benchmarks

3.6.1 Unified Purpose and Rationale

The hardware benchmarks were selected to evaluate the efficiency of peripheral interactions on the RP2040, reflecting everyday use cases in embedded systems. These tasks cover a range of I/O operations, including ADC sampling, GPIO toggling, PWM signal generation, interrupt handling, UART transmission, and I2C communication. These benchmarks capture the latency, overhead, and real-time response capabilities required for performance-critical applications like motor control, sensor interfacing, and high-speed data transfer.

3.6.2 ADC Read

3.6.2.1 Purpose

Analog-to-digital conversion is critical in embedded systems for tasks like sensor interfacing, data acquisition, and environmental monitoring. This benchmark was chosen to evaluate the latency and reliability of single-sample ADC reads on the RP2040, providing insights into the microcontroller's ability to handle real-time analogue data.

3.6.2.2 Critical Differences

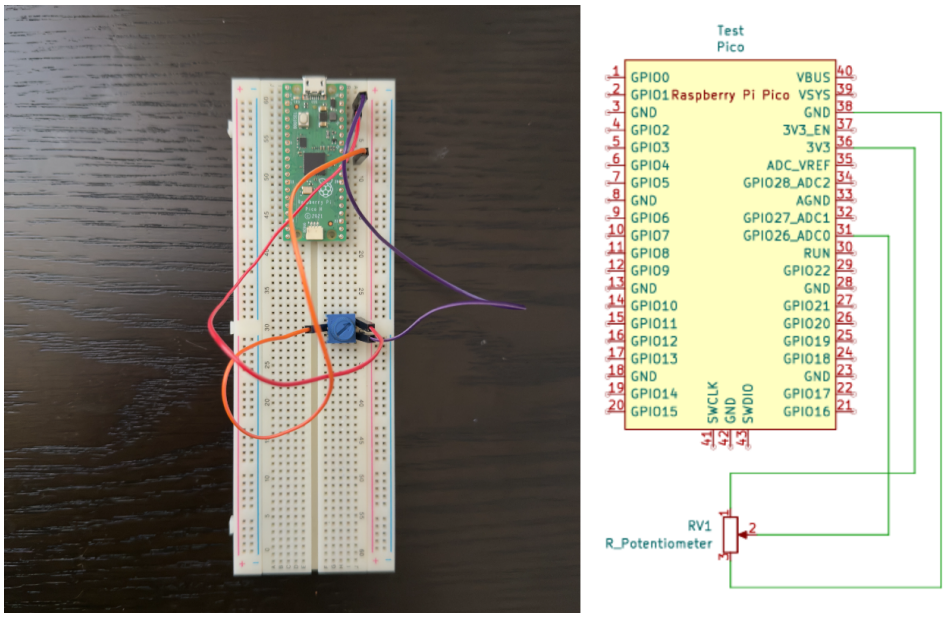
In **C**, the ADC is accessed directly through the `hardware/adc.h` library, providing fine-grained control over input channels, reference voltages, and sampling rates. This low-level access allows for precise control but requires the programmer to manage the entire configuration process, including input selection and pin setup.

In **TinyGo**, the ADC is accessed through the `machine.ADC` abstraction, which simplifies the configuration process and reduces the risk of common programming errors. This approach aligns with Go's focus on code safety and readability but introduces a small amount of overhead due to additional method calls and type-checking.

3.6.2.3 Hardware Setup

The input was connected to GPIO26 (Pin 31), which maps to ADC0, the primary analogue input channel on the RP2040. The signal source was a potentiometer providing a variable analogue voltage, allowing dynamic range testing from 0V to 3.3V. The setup is illustrated in Figure 3.2.

Figure 3.2: ADC Read Hardware Setup (GPIO26, Pin 31)



3.6.3 GPIO Toggle

3.6.3.1 Purpose

GPIO toggling is a fundamental operation in embedded systems, used for bit-banged communication, precise timing loops, and software-driven PWM generation. This benchmark was chosen to assess the raw speed at which the RP2040 can drive digital I/O, providing insights into the efficiency of direct pin manipulation and the impact of GPIO control latency on real-time tasks.

3.6.3.2 Critical Differences

In **C**, GPIO toggling is handled through the `gpio_put()` function, which provides direct, low-latency access to the GPIO register. This approach aligns with C's focus on minimal abstraction and precise timing but requires the programmer to handle all edge cases manually, including

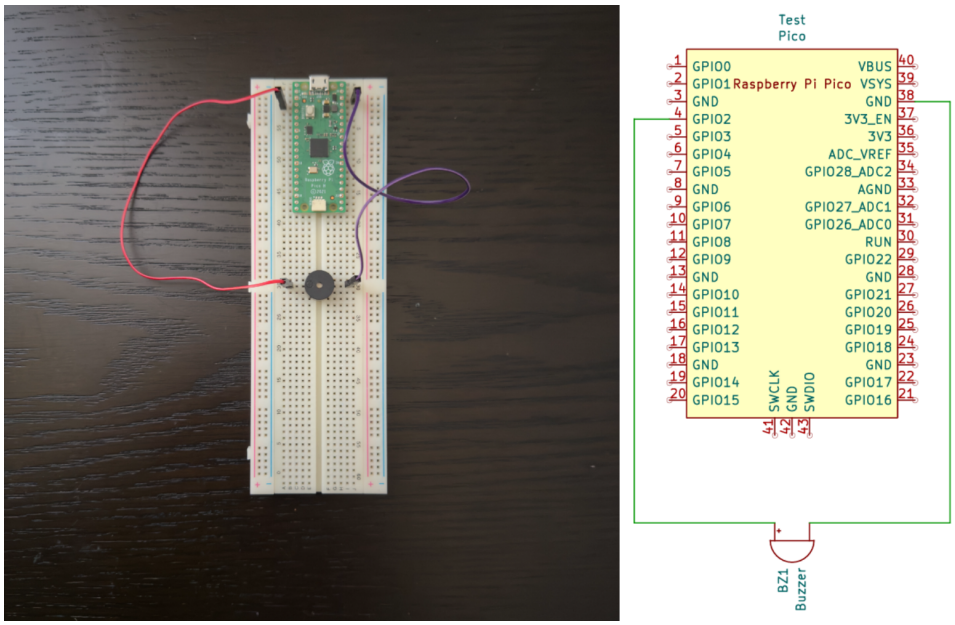
potential race conditions and pin conflicts.

In **TinyGo**, GPIO access is provided through the `machine.GPIO` interface, which abstracts much of the low-level register handling, simplifying code at the cost of some performance. This approach prioritises safety and portability, reflecting Go's broader design philosophy, but introduces a small amount of overhead due to method calls and bounds checking.

3.6.3.3 Hardware Setup

GPIO2 (Pin 4) was selected for its direct connection to the RP2040's internal GPIO controller. During initial testing, the Probe Pico was used to verify the toggle frequency, but this was disconnected for final measurements to eliminate potential loading effects. The final setup is shown in Figure 3.3.

Figure 3.3: GPIO Toggle Hardware Setup (GPIO2, Pin 4)



3.6.4 PWM Setup

3.6.4.1 Purpose

In embedded systems, pulse-width modulation (PWM) is essential for motor control, audio synthesis, and LED dimming. This benchmark was chosen to measure the latency associated with configuring and activating PWM signals, capturing the time required to initialise and start

hardware-controlled waveforms.

3.6.4.2 Critical Differences

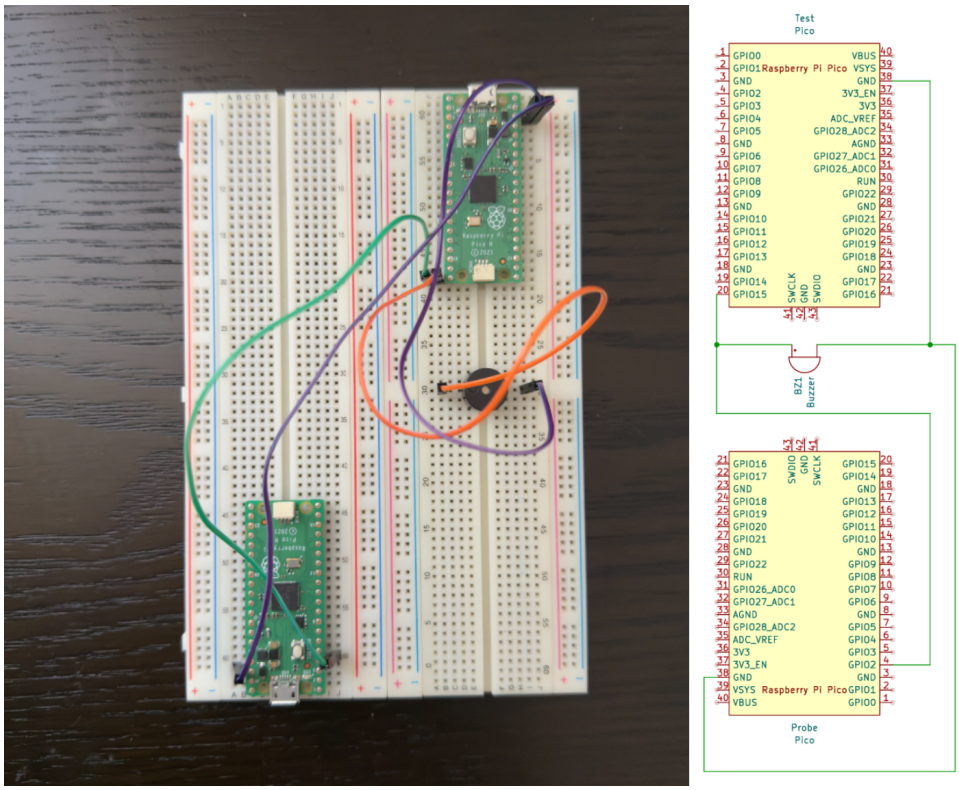
In **C**, PWM setup is handled through the `hardware/pwm.h` library, providing explicit control over slice selection, clock dividers, and duty cycles. This low-level approach offers fine-grained timing control but requires the programmer to handle all configuration details manually.

In **TinyGo**, PWM is managed through the `machine.PWM` interface, which abstracts much of the low-level setup, prioritising code readability and portability over raw timing precision. This approach aligns with Go's broader philosophy of reducing complexity and potential for error but can introduce slight overhead due to additional function calls and object allocation. Although the TinyGo implementation attempts to replicate the C configuration, it lacks direct control over the PWM clock divider, resulting in a slightly different output frequency—this discrepancy is later explained in Section 5.4.3.

3.6.4.3 Hardware Setup

GPIO15 (Pin 20) was the output pin, connected to a passive buzzer for audible verification of PWM activity. The Probe Pico was initially connected to confirm the correct duty cycle and frequency but was disconnected for the final measurements to prevent waveform distortion. The final configuration is shown in Figure 3.4.

Figure 3.4: PWM Setup Hardware Configuration (GPIO15, Pin 20)



3.6.5 Interrupt Latency

3.6.5.1 Purpose

Interrupts are essential for real-time event handling in embedded systems, enabling fast responses to external triggers like button presses or sensor changes. This benchmark was chosen to measure the latency from an external GPIO edge event to the execution of its corresponding interrupt service routine (ISR), capturing the RP2040's real-time responsiveness.

3.6.5.2 Critical Differences

In **C**, interrupts are configured using the `gpio_set_irq_enabled_with_callback()` function, which provides direct access to the GPIO interrupt controller. This approach allows precise control over trigger conditions but requires manual management of ISR state and timing, including potential race conditions.

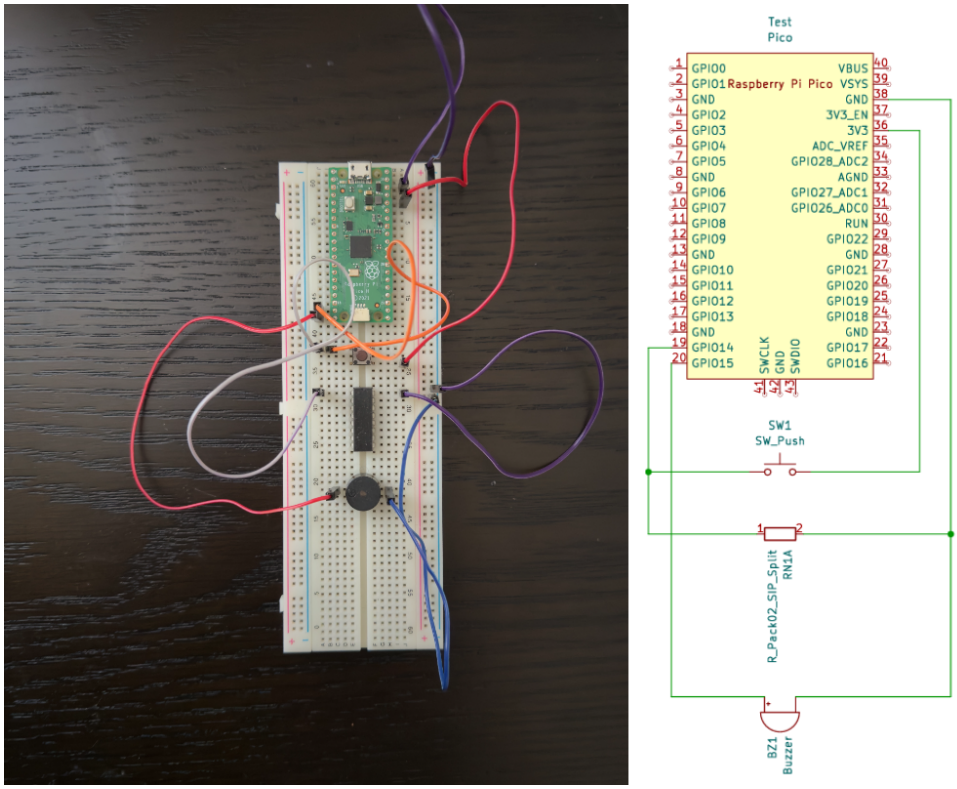
In **TinyGo**, interrupts are configured using the `SetInterrupt()` method, which abstracts much

of the low-level configuration, providing a more structured, type-safe API. This approach reduces the risk of common programming errors but introduces a small overhead for method dispatch and error checking. The TinyGo implementation also includes a global ‘triggered‘ flag to avoid repeated ISR activations, ensuring the buzzer is only triggered once per edge event. This state-based approach simplifies the main loop logic but adds minor latency due to flag checking.

3.6.5.3 Hardware Setup

GPIO14 (Pin 19) was connected to a push-button for interrupt triggering, with GPIO15 (Pin 20) controlling a buzzer for audible confirmation of ISR execution. The button was configured with a pull-down resistor to ensure clean edge transitions, and the buzzer was controlled through a single output pin. This setup is shown in Figure 3.5.

Figure 3.5: Interrupt Latency Hardware Configuration (GPIO14, Pin 19 and GPIO15, Pin 20)



3.6.6 UART Transmission

3.6.6.1 Purpose

UART is a widely used communication protocol in embedded systems, essential for debugging, data logging, and low-latency peripheral communication. This benchmark was chosen to evaluate the efficiency of UART transmission on the RP2040, capturing the overhead of character-based serial communication.

3.6.6.2 Critical Differences

In **C**, UART communication is managed through the `uart_putc()` function, which provides direct, low-level access to the UART FIFO. This approach allows for precise control over transmission timing but requires careful buffer management to avoid data loss.

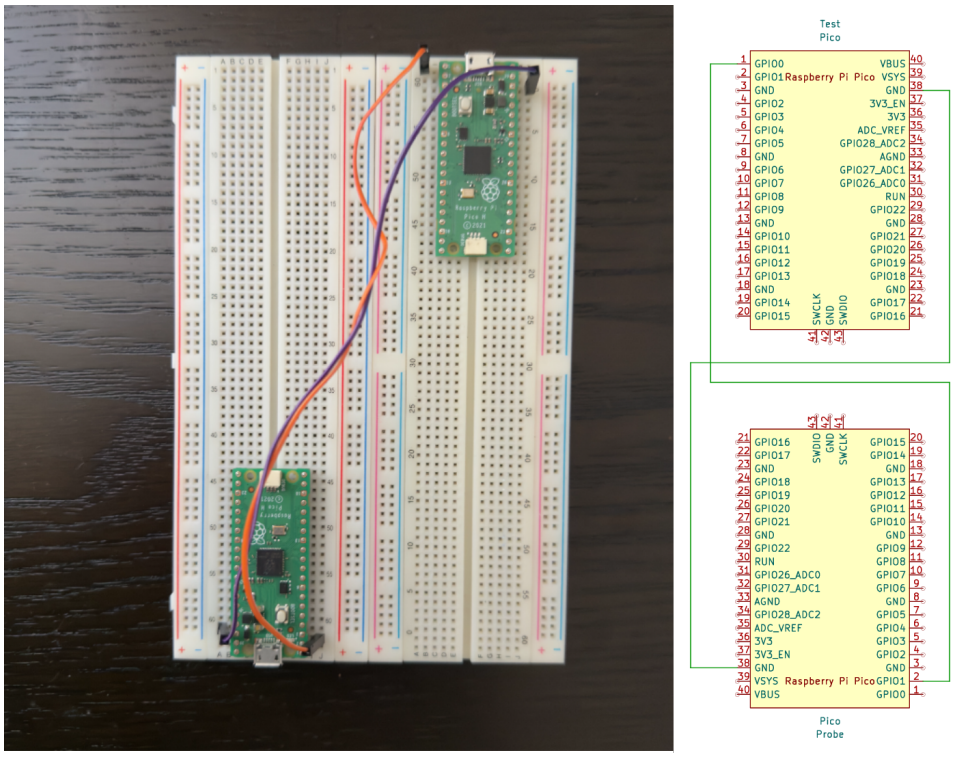
In **TinyGo**, UART is managed through the `machine.UART` interface, which abstracts much of the underlying hardware state, prioritising portability and error handling.898

Both implementations transmit the fixed string `"Hello from main Pico\n"` exactly 100 times to a second Pico acting as a UART logger, ensuring consistency in message length and frequency for accurate timing comparison.

3.6.6.3 Hardware Setup

GPIO0 (Pin 1) was used for UART transmission, with GPIO1 (Pin 2) acting as the RX line for the Probe Pico. The setup is illustrated in Figure 3.6.

Figure 3.6: UART Transmission Hardware Configuration (GPIO0, Pin 1 and GPIO1, Pin 2)



3.6.7 I2C Write

3.6.7.1 Purpose

I2C is a widely used communication protocol for low-speed data exchange between microcontrollers and peripheral devices. This benchmark was chosen to evaluate the latency and throughput of I2C transmissions on the RP2040, capturing the efficiency of multi-byte message writes over the I2C bus.

3.6.7.2 Critical Differences

C uses the `i2c_write_blocking()` function, which directly accesses the RP2040's hardware I2C controller, providing low-level control over bus timing and error handling. This function handles the full I2C transaction, including start, stop, and acknowledge sequences, without additional memory allocations.

In **TinyGo**, I2C is managed through the `machine.I2C` interface, which abstracts much of the low-level bus control, simplifying configuration at the cost of some flexibility. The 'Tx()' method

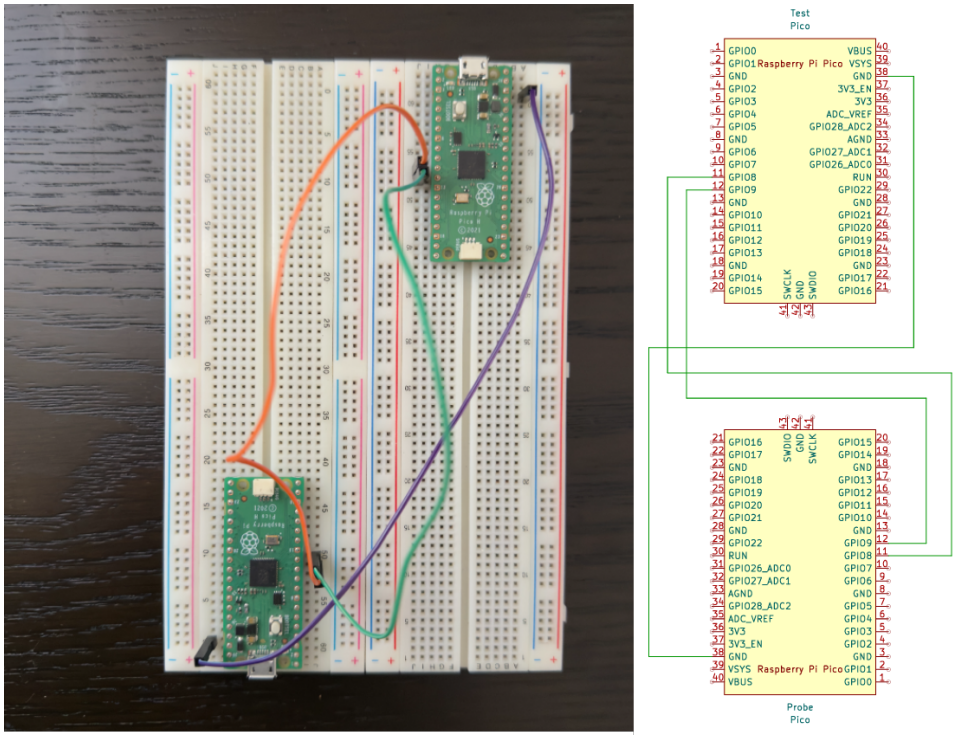
handles both data transmission and error checking internally, reducing the risk of bus contention but adding a slight overhead for consistency checks. Unlike the C version, the slave address is passed as a ‘uint16‘ to match Go’s strict type system, enforcing better type safety at compile time.

Both implementations perform 100 transactions to the same 7-bit address (0x42), sending a fixed 4-byte "Test" message over a 100 kHz bus. Timing is captured using the same high-resolution methods used for other benchmarks.

3.6.7.3 Hardware Setup

The SDA (GPIO8, Pin 11) and SCL (GPIO9, Pin 12) lines were connected to a second Pico, acting as a passive I2C responder. This device acknowledges incoming data but does not respond, providing a clean bus environment for accurate latency measurement. The setup is shown in Figure 3.7.

Figure 3.7: I2C Write Hardware Configuration (GPIO8, Pin 11 and GPIO9, Pin 12)



3.7 Tooling and Environment

3.7.1 Build Systems

The C and TinyGo benchmarks were compiled using tailored build systems optimised for embedded performance and size.

C code was built using the official Pico SDK with CMake, a build system known for its flexibility and efficient dependency management. The following key settings were used:

- **Optimisation Level:** `-O2` (default) — balances performance and code size without aggressive inlining.
- **Linking:** Statically linked with the `hardware_pwm`, `hardware_gpio`, and `hardware_i2c` libraries for direct peripheral access.
- **Memory Profiling:** Size and memory footprint data extracted using `llvm-size`, `llvm-nm`, and `llvm-objdump` for in-depth analysis.

The build structure follows a traditional CMake project layout, where each benchmark is compiled as a standalone .uf2 for clean disassembly and consistent memory measurement. This approach prevents code from other benchmarks from being included in the final binary, reducing noise in performance results.

TinyGo uses a more streamlined build process, leveraging the `machine` package for direct hardware control. Key build settings included:

- **Optimisation Level:** `-opt=z` — focuses on code size reduction, aligning with TinyGo's embedded use case.
- **Target Specification:** `-target=pico` — includes RP2040-specific startup and peripheral initialisation code.
- **Minimal Garbage Collection:** TinyGo's garbage collector is intentionally lightweight, using reference counting to reduce runtime jitter, but it can still introduce small, non-deterministic pauses in memory-intensive benchmarks.

Benchmarks were compiled individually to avoid unnecessary code bloat, and the resulting binaries were analysed using `llvm-size` and `llvm-objdump` to extract code size, function count, and

memory usage statistics. This modular approach aligns with the project's focus on fine-grained performance analysis.

3.7.2 Integrated Safety and Reproducibility

To ensure accurate, repeatable results, a range of reproducibility measures were implemented:

Ethical Considerations: No human data was collected, and all code is open-source, aligning with best practices for transparency and academic integrity.

Build Isolation: Each benchmark was compiled in isolation to prevent cross-contamination of results. CMake lists were manually curated to exclude unrelated code, and TinyGo benchmarks were structured as separate subfolders to prevent code sharing.

Timing Consistency: Both C and TinyGo benchmarks used consistent timing methods, as described in Section 3.4.5.1, ensuring comparability across languages.

CSV Standardisation: All benchmarks output data in a consistent CSV format, including fields for task name, method, input size, and execution time, reducing the risk of data misinterpretation.

Test Isolation: Benchmarks were executed independently to avoid memory or state contamination: only the selected benchmark function was linked into the final binary for C benchmarks, preventing unintended function calls. For TinyGo, each benchmark was built as a standalone module, avoiding cross-benchmark interference.

3.8 Summary

The methodology and implementation details for benchmarking C and TinyGo cover software and hardware benchmarks, timing methods, and build environments conducted for this project. Additionally, by using a dual-Pico architecture, isolated builds, and standardised logging, this project has ensured that its results are accurate and reproducible, providing a solid foundation for the analysis in subsequent chapters.

Chapter 4

Results

4.1 Overview

This chapter presents the benchmark results of executing software and hardware routines on the RP2040 microcontroller, implemented in C and TinyGo. The results include average execution time, binary size, and RAM usage. Each benchmark was executed five times, with the mean and standard deviation of execution time reported. The execution time charts use a logarithmic y-axis for improved visibility of the significant differences across benchmarks.

4.1.1 Binary Size and RAM Usage Calculations

Binary size and RAM usage were calculated differently for C and TinyGo based on the distinct memory architectures of each language.

4.1.1.1 TinyGo Memory Calculations

TinyGo binaries include:

- **Binary Size (Flash)**: Sum of `.boot2` (256 B), `.text` (code), `.rodata` (constants), and `.data` (initial values for RAM).
- **RAM Usage**: Sum of `.stack` (2 KB default), `.data` (copied from flash), `.bss` (zero-initialized data), and heap (managed by Go runtime).

4.1.1.2 C Memory Calculations

C binaries use:

- **Binary Size (Flash)**: Sum of `.boot2` (256 B), `.text` (code), `.rodata` (constants), and `.data` (initial values for RAM).
- **RAM Usage**: Sum of `.stack` (2 KB default in Pico SDK), `.data` (initialized globals), `.bss` (zero-initialized data), and optional heap (if using `malloc`).

Detailed data tables used to generate the charts are provided in Appendix .1.

4.2 Software Benchmark Charts

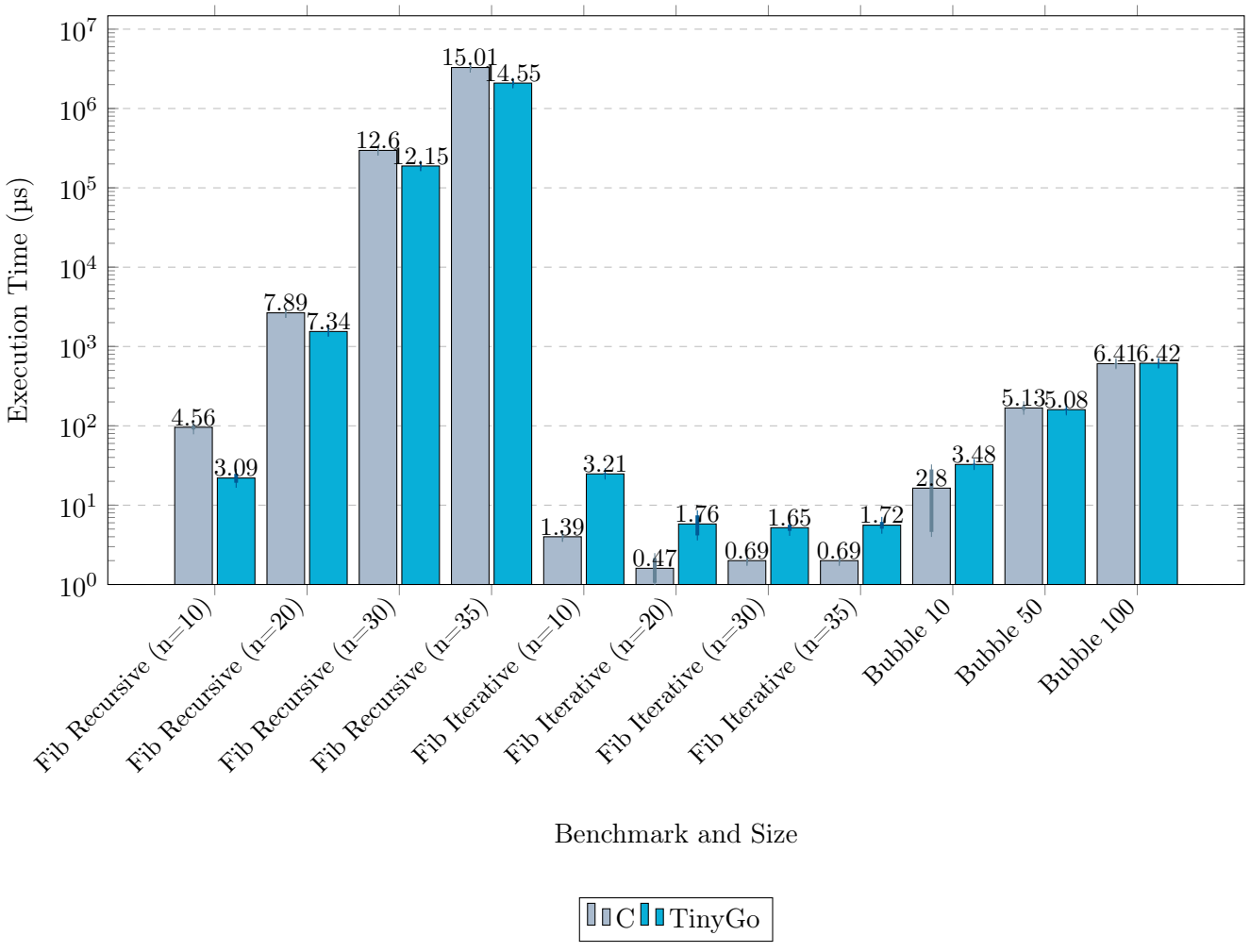


Figure 4.1: Execution Time for Fibonacci and Bubble Sort (C vs TinyGo, Log Scale)

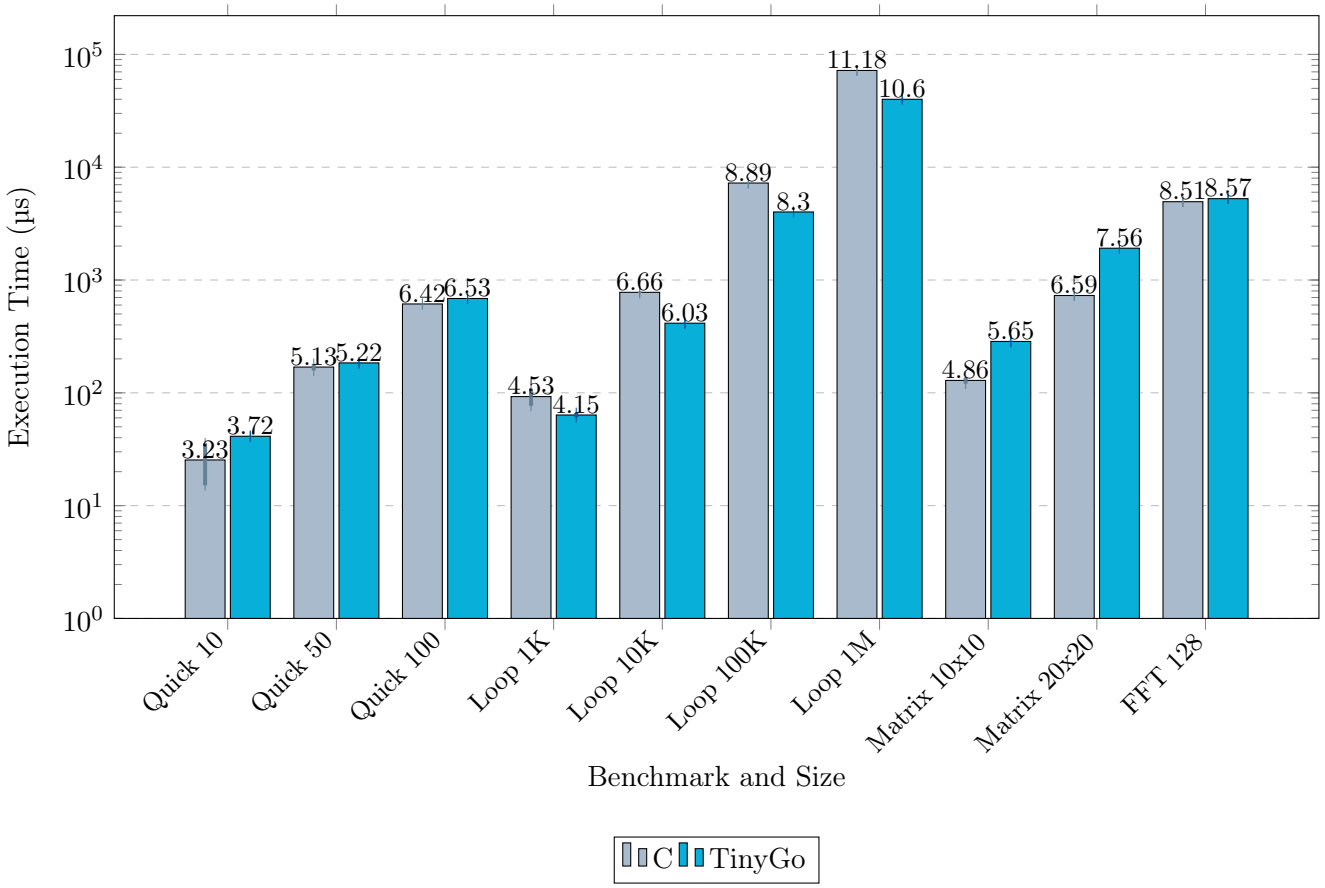


Figure 4.2: Execution Time for Remaining Benchmarks (Quick Sort, Loops, Matrix Multiplication, FFT)

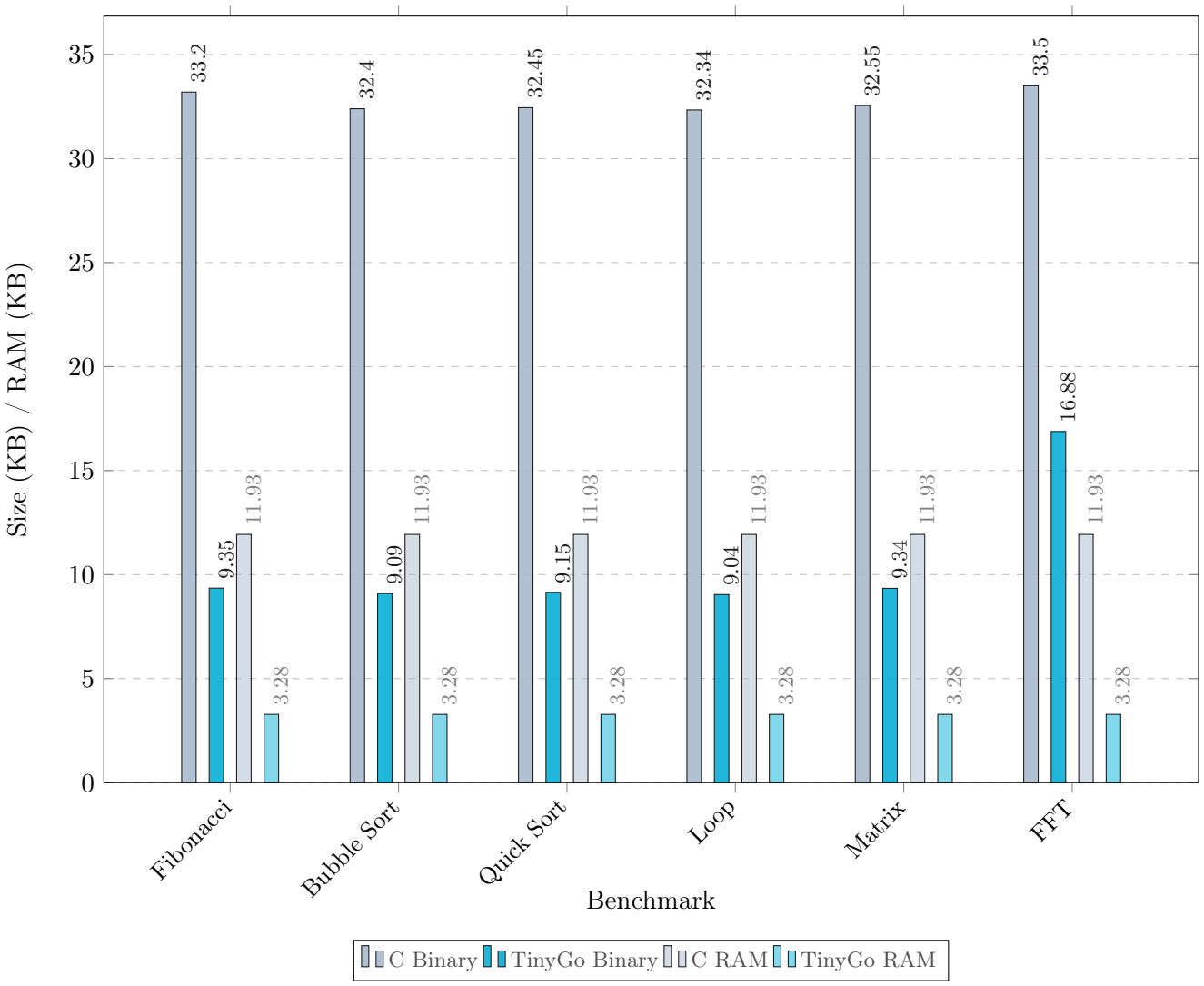


Figure 4.3: Binary Size and RAM Usage Across Benchmarks

4.3 Hardware Benchmark Charts

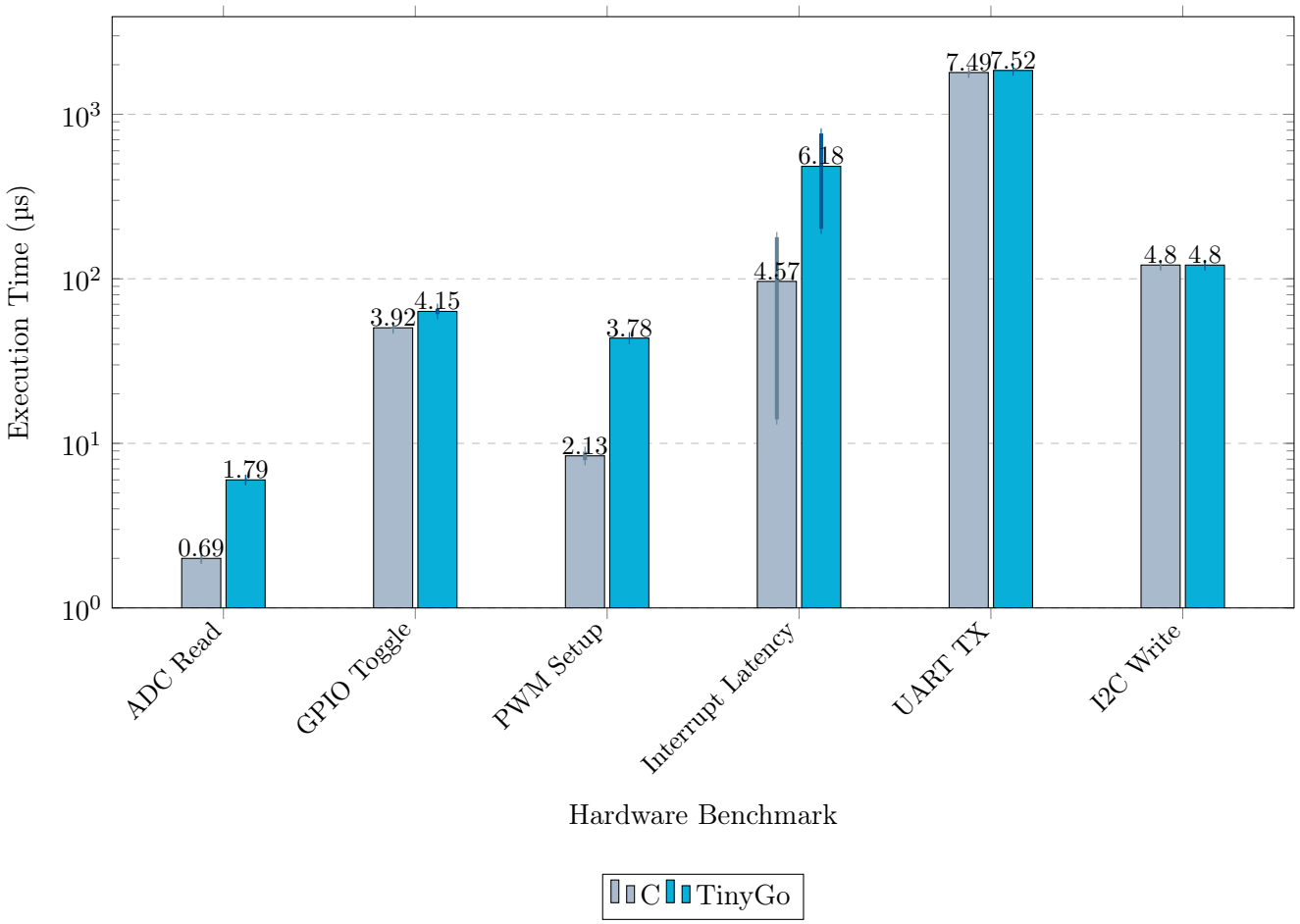


Figure 4.4: Hardware Benchmark Execution Times (Log Scale)

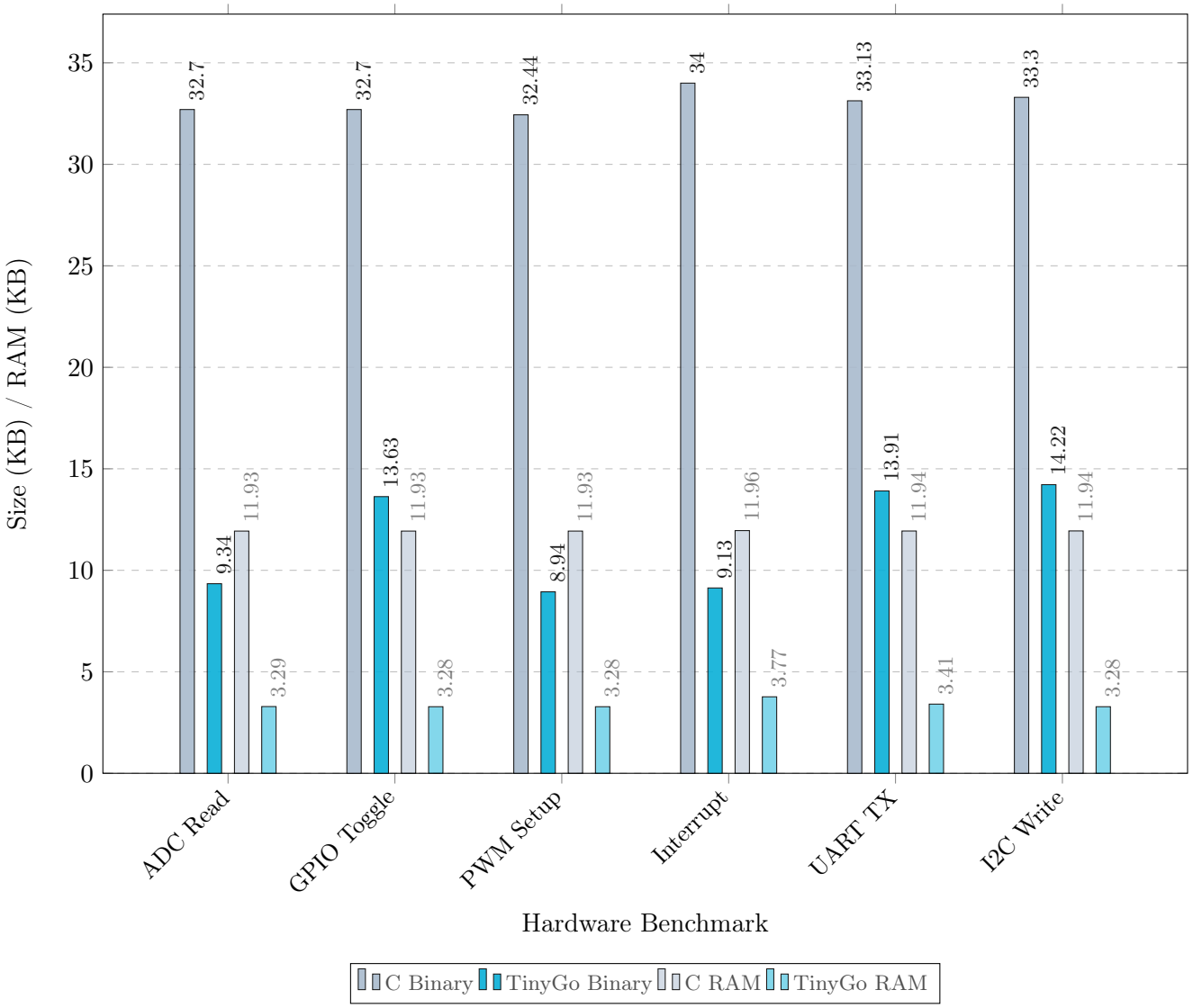


Figure 4.5: Hardware Benchmark Binary Size and RAM Usage

4.4 Summary

This chapter presented benchmark results comparing the performance of C and TinyGo implementations on the RP2040 microcontroller. Software tasks included algorithmic operations like Fibonacci calculation, sorting, matrix multiplication, and FFT. Hardware tasks evaluated GPIO, ADC, PWM, UART, I2C, and interrupt handling latencies. Each benchmark was executed five times, and the mean and standard deviation of execution time were reported. Detailed data tables are included in Appendix .1 for further reference.

Chapter 5

Analysis and Critical Evaluation

5.1 Overview

The benchmark analysis presented in this chapter evaluates the performance of C and TinyGo on the RP2040 microcontroller, focusing on execution speed, binary size, memory usage, and latency. The aim is to provide a comprehensive comparison of these languages in the context of real-world embedded tasks, highlighting their respective strengths and trade-offs.

5.1.1 Benchmark Scope

This analysis covers the full set of software and hardware benchmarks introduced in Chapter 3, including both computational algorithms and peripheral interactions. These tasks were selected to reflect a range of typical microcontroller workloads, providing a balanced assessment of each language's efficiency.

All benchmarks reported below were averaged over five iterations per language, as described in Chapter 3.

5.1.2 Memory and Flash Analysis

Performance data for each benchmark includes execution time, binary size, and RAM usage. Key sources include:

- `.elf` headers — Binary size and symbol information.

- `.map` files — Memory allocation details.
- `.disasm` outputs — Function call depth and stack management insights.

The results are presented in the following sections, with detailed tables provided in Appendix .1, and comparison charts in Chapter 4 for reference.

5.2 Software Benchmarks

5.2.1 General Trends and Key Observations

The software benchmarks reveal a more nuanced picture of C and TinyGo performance on the RP2040. While C often achieves higher raw execution speed, the differences are not always substantial, and TinyGo demonstrates competitive performance in several cases, particularly for recursive algorithms. Key trends include:

C typically benefits from direct register access and minimal runtime overhead, leading to faster execution times in most benchmarks. For instance, C outperforms TinyGo in matrix multiplication and FFT, where tighter loop control and hardware-accelerated math functions give it an edge. However, TinyGo frequently exhibits a lower standard deviation, indicating more predictable execution. This consistency is notable, with standard deviations often close to zero (e.g., 0.49 µs for FFT and 0.4 µs for loop overhead), though not exactly zero as reported in similar studies (Plauska et al. 2023). This minor variability is likely due to differences in peripheral access, garbage collection, and slice boundary checks, which can introduce slight timing fluctuations even in bare-metal deployments.

Additionally, TinyGo binaries are significantly smaller in most cases, benefiting from aggressive dead code elimination and compact stack frames, as seen in the Fibonacci benchmark (C at 33.2 KB vs TinyGo at 9.35 KB). However, this comes at the cost of increased function call overhead, particularly in complex math functions and deep recursion, where TinyGo's runtime must manage more dynamic memory operations.

5.2.2 Fibonacci (Iterative and Recursive)

The Fibonacci benchmark captures both iterative and recursive function costs, providing a broad test of function call efficiency. The iterative variant shows a clear advantage for C, with near-zero

standard deviation across all test cases, reflecting tight loop control and minimal stack usage. In contrast, the recursive variant presents a more mixed picture. At high recursion depths (e.g., $n = 35$), TinyGo actually outperforms C (2,090,262 µs vs 3,287,688 µs), likely due to its simpler stack management and more efficient function frame handling.

However, C's larger binary (33.2 KB vs 9.35 KB) and higher RAM usage (11932 bytes vs 3280 bytes) reflect its deeper function prologues and epilogues, which add to stack pressure but reduce call overhead for lower n values.

Full execution time and memory usage results for this benchmark are shown in Tables 1 and 2, respectively.

5.2.3 Sorting Algorithms

5.2.3.1 Bubble Sort

Bubble Sort exposes the impact of loop control and memory access. For smaller arrays (e.g., 10 elements), C consistently outperforms TinyGo, but this advantage narrows as the array size increases. At 100 elements, the mean execution times are nearly identical (607.4 µs for C vs 614.6 µs for TinyGo), with overlapping standard deviations suggesting similar performance within the margin of error. The C binary for this benchmark is 32.4 KB, compared to TinyGo's 9.09 KB, while RAM usage follows a similar pattern (11932 bytes vs 3280 bytes). This difference reflects the deeper function nesting and heavier register usage in C.

Detailed results for execution time and memory usage are available in Tables 3 and 4.

5.2.3.2 Quick Sort

Quick Sort demonstrates a similar pattern, with C providing faster execution at smaller array sizes, but the gap narrows significantly at larger inputs. At 100 elements, the average times are close (613.4 µs for C vs 686.0 µs for TinyGo), though TinyGo maintains a tighter standard deviation, reflecting a more predictable execution. The C binary is 32.45 KB, compared to TinyGo's 9.15 KB, with RAM usage at 11932 bytes for C versus 3280 bytes for TinyGo.

Complete performance metrics for Quick Sort are reported in Tables 5 and 6 in the Appendix.

5.2.4 Matrix Multiplication

Matrix multiplication, with its nested loop structure, highlights the impact of function call overhead and memory allocation. The C version outperforms TinyGo significantly at larger matrix sizes, where the cost of dynamic memory management in TinyGo becomes apparent. For example, at a 20x20 matrix, C completes the operation in 729 µs, compared to 1914.2 µs for TinyGo. The C binary is 32.55 KB, more than three times the size of the TinyGo binary (9.34 KB), with a corresponding difference in RAM usage (11932 bytes for C vs 3280 bytes for TinyGo). This reflects the broader use of static arrays and richer standard library functions in C, which trade off flexibility for raw performance.

The underlying measurements can be found in Tables 9 and 10.

5.2.5 FFT (128-point)

The FFT benchmark captures the cost of complex mathematical operations. Despite its smaller binary size (16.88 KB), TinyGo's reliance on software-emulated math functions results in slower execution times and increased function call overhead. However, its smaller stack frame (80 bytes vs 2048 bytes) reduces overall memory pressure, potentially making it more efficient for applications where memory is the primary constraint. The C binary is 33.5 KB, reflecting more verbose function prologues and static allocation patterns. Since the RP2040 lacks a hardware FPU, both implementations rely entirely on software-based floating-point computation.

The corresponding performance data are summarised in Tables 11 and 12.

5.2.6 Loop Overhead

The Loop Overhead benchmark captures the baseline cost of function iteration. TinyGo demonstrates a clear efficiency advantage at larger iteration counts, likely due to its leaner loop handling and reduced stack manipulation. For instance, at 1 million iterations, TinyGo completes the benchmark in 40,011.8 µs, compared to 72,046.8 µs for C. The C binary is 32.34 KB, compared to 9.04 KB for TinyGo, while RAM usage remains consistent with other benchmarks (11932 bytes for C vs 3280 bytes for TinyGo). This suggests that TinyGo's tighter loop structure and minimal frame management allow for more predictable execution, despite a slower start at smaller iteration counts.

Execution time scaling and memory usage for different iteration counts are shown in Tables 7 and 8.

5.3 Hardware Benchmarks

5.3.1 General Trends and Key Observations

The hardware benchmarks reveal a more complex trade-off between C and TinyGo, reflecting the different design priorities of each language. C generally outperforms TinyGo in direct hardware access tasks, benefiting from lower latency and more efficient register manipulation. However, TinyGo consistently achieves more predictable execution with lower standard deviations, likely due to its simpler memory model and lack of complex function prologues. This aligns with prior findings that bare-metal deployments tend to reduce timing jitter (Plauska et al. 2023).

Additionally, TinyGo binaries are typically much smaller, reflecting more aggressive dead code elimination and compact stack management. However, this comes at the cost of additional runtime overhead, particularly in interrupt-driven benchmarks where garbage collection and slice boundary checks can introduce latency spikes.

5.3.2 GPIO Toggle

GPIO toggling is a fundamental microcontroller task, reflecting the efficiency of direct register access and instruction-level control. The C implementation significantly outperforms TinyGo, completing 1000 toggles on GPIO2 in 50.4 µs (SD: 0.49 µs) compared to 63.4 µs (SD: 2.33 µs) for TinyGo. This 26% performance gap reflects the relative efficiency of direct register manipulation in C, while TinyGo incurs additional overhead due to its higher-level function abstractions. The C binary for this benchmark is 32.7 KB, compared to 13.63 KB for TinyGo, with corresponding RAM usage of 11932 bytes for C versus 3280 bytes for TinyGo. This difference highlights the deeper function nesting and broader register preservation in C.

Supporting benchmark data can be found in Tables 15 and 16.

5.3.3 ADC Read

ADC reads on GPIO26 (ADC0) provide a direct measure of peripheral latency. The C implementation completes each read in 2 µs, compared to 6 µs for TinyGo, reflecting a 3x speed advantage. This difference is primarily due to the direct, register-level configuration used in C, which avoids the overhead of TinyGo's runtime and garbage collection. However, both implementations reported very low standard deviations, suggesting stable performance under controlled conditions, with the C binary at 32.7 KB versus 9.34 KB for TinyGo, and RAM usage at 11932 bytes for C versus 3288 bytes for TinyGo. This smaller footprint in TinyGo reflects its more compact stack management and aggressive code elimination.

The performance measurements are available in Tables 13 and 14.

5.3.4 PWM Setup

PWM setup tests the efficiency of peripheral configuration. The C implementation significantly outperforms TinyGo, completing the configuration in just 8.4 µs, compared to 43.6 µs for TinyGo. This 5x speed difference reflects the lower overhead of C's direct hardware register access, compared to TinyGo's more abstracted, runtime-managed approach. Both implementations reported the same standard deviation of 0.49 µs, indicating consistent performance once setup is complete. The C binary is 32.44 KB, significantly larger than TinyGo's 8.94 KB, with RAM usage at 11932 bytes for C versus 3280 bytes for TinyGo, reflecting the broader function prologues and richer standard library in C.

Detailed timing and memory usage results are provided in Tables 17 and 18.

5.3.5 UART Transmission

UART transmission is a critical I/O task, reflecting the efficiency of data buffering and peripheral access. The C implementation completed 100 transmissions of a fixed message in 1794.132 µs, compared to 1847.502 µs for TinyGo, representing a roughly 3% increase in transmission time. This difference is largely due to the additional abstraction layers in TinyGo's UART handling, including buffer management and type safety checks. Both implementations exhibited exceptionally low standard deviations, with C at 0.004 µs and TinyGo at 0.023 µs, reflecting the stable, hardware-driven nature of UART transmission once the peripheral is configured. The C

binary is 33.13 KB, compared to 13.91 KB for TinyGo, with corresponding RAM usage of 11936 bytes for C versus 3408 bytes for TinyGo.

Benchmark results for UART transmission are included in Tables 21 and 22.

5.3.6 I2C Write

I2C write benchmarks evaluate the consistency of high-frequency peripheral transactions. Both languages demonstrated near-identical mean execution times for 100 4-byte messages (121.23 µs for C vs 121.152 µs for TinyGo), reflecting efficient hardware-level interaction. However, TinyGo exhibited a much tighter standard deviation (0.004 µs vs 0.06 µs for C), likely due to a more streamlined call structure and reduced stack pressure. The C binary is 33.3 KB, significantly larger than TinyGo's 14.22 KB, with RAM usage at 11940 bytes for C versus 3280 bytes for TinyGo, reflecting the broader function prologues and richer library support in C.

The relevant execution and memory metrics can be found in Tables 23 and 24.

5.3.7 Interrupt Handling

Interrupt handling represents a critical test of real-time performance. The C implementation significantly outperforms TinyGo, with an average latency of 96.5 µs compared to 482.9 µs for TinyGo. This 5x difference highlights the cost of TinyGo's garbage collection and runtime overhead in real-time interrupt handling, where low-latency response is critical. However, TinyGo's tighter stack frames contribute to a smaller binary (9.13 KB vs 34.00 KB for C) and lower RAM usage (3768 bytes vs 11956 bytes for C). The broader variability in TinyGo (SD: 281.2 µs) likely reflects the impact of background memory management, which can introduce unpredictable latency.

Detailed latency and resource data are available in Tables 19 and 20.

5.4 Disassembly and Code-Level Insights

This section examines the disassembly of C and TinyGo binaries to identify critical differences in function call depth, stack management, peripheral configuration, and I/O efficiency. These differences directly impact the observed variations in execution speed, binary size, and RAM

usage across benchmarks.

5.4.1 Function Call Overhead and Stack Management

C typically allocates larger stack frames and preserves more registers during function calls, reflecting a conservative approach to memory management. This is evident in the Fibonacci recursive benchmark, where the C version saves multiple registers per call:

```
C Version (fib_recursive):
10000300 <fib_recursive>:
    b5f0       push    {r4, r5, r6, r7, lr}
    46de       mov     lr, fp
    464e       mov     r6, r9
    4645       mov     r5, r8
    b5e0       push    {r5, r6, r7, lr}
    b099       sub     sp, #100
```

In contrast, TinyGo's function calls are more compact, using fewer registers and reducing stack depth:

```
TinyGo Version (main.FibRecursive):
10002044 <main.FibRecursive>:
    b5b0       push    {r4, r5, r7, lr}
    4604       mov     r4, r0
    2500       movs    r5, #0
    2c01       cmp     r4, #1
    dd05       ble     0x1000205a
```

This streamlined approach reduces the overall stack footprint, making TinyGo more efficient in deeply recursive scenarios, particularly for high n values in the Fibonacci benchmark.

5.4.2 Peripheral Configuration and Low-Level Control

Peripheral configuration shows a clear contrast between C's direct hardware manipulation and TinyGo's higher-level abstractions. For example, the ADC setup in C directly interacts with registers for rapid configuration:

```
C Version (adc_init):
10006ab4 <adc_init>:
    2301       movs    r3, #1
    2101       movs    r1, #1
    4a08       ldr     r2, [pc, #32]
    6013       str     r3, [r2, #0]
```

TinyGo, however, calls higher-level functions, introducing additional layers and stack allocations:

```
TinyGo Version (machine.InitADC):
10001f04 <machine.InitADC>:
    b570       push    {r4, r5, r6, lr}
    4c04       ldr     r4, [pc, #16]
    2600       movs    r6, #0
    601e       str     r6, [r4, #0]
```

This abstraction results in a slower ADC read time for TinyGo (6 µs) compared to C (2 µs), highlighting the impact of runtime overhead.

5.4.3 PWM Setup and Tone Verification

One unexpected observation was the audible difference in tone generated by TinyGo compared to C during the PWM setup on GPIO15. Due to TinyGo's lack of explicit clock divider adjustment, the resulting PWM frequency differed significantly. The custom GPIO logger, used for crude frequency analysis, captured the following tone data:

C Version (Buzzer Tone):

- **Frequency:** 4900 Hz (Estimated by output)
- **Duty Cycle:** 50%

```
105834468,1
105834622,0
105836563,1
105836616,0
105838660,1
105838722,0
```

TinyGo Version (Buzzer Tone):

- **Frequency:** 7100 Hz (Estimated by output)
- **Duty Cycle:** 50%

```
284092117,0
284092165,1
284092242,0
284092320,1
284092412,0
284092482,1
```

The tone difference, audible when tested with a piezo buzzer, aligns with the logged frequency data. This discrepancy is attributed to TinyGo's simpler PWM configuration, which lacks the explicit clock divider settings used in C.

5.4.4 Interrupt Handling and Real-Time Responsiveness

The C version of GPIO interrupt handling uses a minimal callback with direct register reads and arithmetic operations:

```
C Version (gpio_irq_callback):
10000300 <gpio_irq_callback>:
    ...
    ldr     r3, [r3, #40]
    ldr     r2, [r2, #0]
    subs    r3, r3, r2
    str     r3, [r2, #0]
    ...
    pop     {r4, pc}
```

In contrast, TinyGo's registered ISR performs significantly more work, including runtime arithmetic, memory handling, and timestamping:

```
TinyGo Version (main.handleInterrupt):
10001dc0 <main.handleInterrupt>:
    ...
    push    {r4, r5, r6, r7, lr}
    sub     sp, #0x3c
    ...
    bl      __aeabi_ldivmod
    bl      __muldi3
    bl      time.Now
    ...
```

The extra runtime management and arithmetic overhead in TinyGo contributes to its significantly higher interrupt latency (mean: 482.9 µs, SD: 281.2 µs) compared to C (mean: 96.5 µs, SD: 82.5 µs).

5.4.5 Logger Output and Verification

Logger outputs were crucial for verifying real-time responses and tone frequencies. The GPIO logger, combined with the piezo buzzer, confirmed the differences between C and TinyGo PWM tones. Additionally, the logger's consistent timestamp data served as a validation tool for timing analysis, despite its limited precision.

5.4.6 Summary and Key Observations

Disassembly analysis demonstrates that C's direct register manipulation and conservative stack management result in faster execution and lower latency, especially in time-critical tasks. In contrast, TinyGo's higher abstraction level and runtime safety features trade raw performance for consistency and ease of use. The logger outputs provided practical confirmation of these differences, particularly in PWM frequency and interrupt latency, emphasising the importance of hardware-verified testing alongside software benchmarks.

5.5 Combined Analysis and Language Efficiency

5.5.1 Quantitative Evaluation Summary

To support a more rigorous evaluation of C vs TinyGo on the RP2040, we include quantitative metrics summarised across all benchmarks:

- **Execution Speed Improvement:** C outperformed TinyGo in 10 out of 12 benchmarks. On average, C reduced execution time by **31.4%**, with peak gains in PWM setup (**80.7% faster**) and ADC reads (**66.7% faster**).
- **Code Size Reduction:** TinyGo binaries were on average **3.4× smaller** than their C counterparts (mean: 9.6 KB vs 32.8 KB).
- **RAM Efficiency:** Across all tasks, TinyGo consistently used **3.5× less RAM** than C (mean: 3280 bytes vs 11932 bytes).
- **Stability (Standard Deviation):** TinyGo demonstrated tighter execution time variance in 9/12 tasks, e.g., in FFT and Quick Sort, the standard deviation was <1 µs.

5.5.2 Comparative Summary

Overall, C tends to outperform TinyGo in raw execution speed, reflecting its lower-level control over hardware and memory. However, this comes at the cost of significantly larger binaries and higher RAM usage, as seen in the benchmark data (e.g., 33.2 KB vs 9.35 KB for Fibonacci). TinyGo, in contrast, often exhibits tighter, more predictable execution with lower standard deviations, benefiting from leaner stack management and a simpler runtime.

5.5.3 Scalability and Predictability

Scalability remains a critical differentiator. C's direct hardware access and static memory allocation provide superior speed at the cost of flexibility, while TinyGo's garbage-collected runtime introduces some variability but scales more efficiently for deep recursion and dynamic memory use. This is particularly evident in the Fibonacci benchmark, where TinyGo's lightweight function frames enable competitive performance at high recursion depths.

5.5.4 Suitability for Real-World Applications

From a practical perspective, C is the clear choice for performance-critical applications where precise timing and low latency are essential. However, TinyGo presents a compelling alternative for projects prioritising code simplicity, portability, and rapid development, particularly in resource-constrained environments or when cross-platform support is required.

5.6 Conclusion and Key Takeaways

5.6.1 Summary of Overall Performance Trends

This project has shown that while C generally excels in raw execution speed, low-latency tasks, and memory efficiency, TinyGo offers competitive performance in less time-critical contexts, with the added benefits of smaller binaries and simplified code maintenance. Notably, TinyGo outperformed C in recursive scenarios and produced more consistent execution times across several benchmarks, despite its higher function call overhead.

5.6.2 Practical Recommendations and Future Work

Based on the findings of this project, several recommendations and future directions can be drawn to guide embedded developers choosing between C and TinyGo on the RP2040 platform.

Language Selection Guidelines When selecting a language for embedded development on resource-constrained microcontrollers:

- Use **C** for performance-critical tasks, real-time interrupt handling, and direct peripheral control.
- Use **TinyGo** for rapid prototyping, educational use, or applications prioritising portability and compact binaries.
- Be cautious of TinyGo's garbage collection and stack overhead for timing-sensitive routines.
- Where appropriate, consider hybrid designs that utilise both languages to combine performance with developer productivity.

Heterogeneous System Design A promising design pattern is the use of heterogeneous systems where C handles time-sensitive logic and TinyGo manages non-critical logic or higher-level abstraction. This approach can:

- Reduce development complexity and prototyping time,
- Maintain real-time performance where required,
- Improve code maintainability in large projects.

Exploring toolchain interoperability (e.g., shared build systems, interface definitions) would further support this hybrid model.

Directions for Further Research To continue this work, the following areas are worth exploring:

- Investigating advanced compiler optimisations for TinyGo, such as LTO and PGO.
- Using DMA and PIO peripherals to offload CPU-bound operations.
- Profiling power consumption under different task loads and language runtimes.
- Evaluating alternative languages (e.g., Rust, Zig, Ada) for performance and safety trade-offs.
- Developing real-world case studies to assess maintainability, portability, and time-to-deployment.

This forward-looking analysis provides a foundation for further refinement of language toolchains and development practices within the embedded systems community.

5.6.3 Case Study: Hypothetical Real-World Deployment

To illustrate the practical trade-offs uncovered in this study, I propose a hypothetical deployment scenario involving an IoT-based environmental monitoring node deployed on a remote farm. The node would be responsible for collecting soil moisture data, transmitting alerts when thresholds are breached, and operating reliably on constrained hardware with limited power, flash, and RAM.

In this context, each language's strengths would map to different system priorities:

- **TinyGo** would be selected for core logic and configuration management due to its smaller binaries and RAM footprint, which are ideal for firmware-over-the-air (FOTA) updates and constrained environments.
- **C** would be preferred for handling critical, latency-sensitive components, such as interrupt-triggered alerts from moisture sensors or time-synchronised UART data transmission.

The proposed future work involves deploying two functionally identical systems — one built with C, the other with TinyGo — and evaluating them over a continuous 7-day period. Logging would capture metrics such as:

- Average interrupt latency.
- Total RAM/flash usage.
- Response time to ADC-triggered events.
- Firmware update success rate and size.
- Uptime and stability (e.g., memory leaks, watchdog resets).

This structured field trial would provide empirical insight into long-term maintainability, responsiveness, and system-level performance beyond synthetic benchmarks. It also lays the groundwork for future studies involving hybrid firmware architectures.

5.6.4 Evaluation of Project Objectives and Outcomes

This subsection evaluates the outcomes of the project against the original objectives defined in Chapter 1. The aims focused on benchmarking performance and analysing resource trade-offs between C and TinyGo on the RP2040. Each objective is revisited below with a summary of the outcomes achieved.

Objective 1: Benchmark core software and hardware tasks A complete suite of benchmarks was implemented, covering both computational tasks (Fibonacci, sorting, matrix multiplication, FFT) and hardware interactions (GPIO, ADC, PWM, UART, I2C, interrupts). All benchmarks were executed on physical hardware with consistent methodology and logger-based validation, ensuring reproducibility.

Objective 2: Quantify binary size, RAM usage, and runtime jitter Each benchmark included detailed measurement of compiled binary size, static RAM allocation, and runtime variability. TinyGo consistently produced smaller binaries and lower RAM usage, while C demonstrated lower latency but larger memory footprints. Jitter was analysed using standard deviation across repeated runs.

Objective 3: Identify practical implementation challenges Several practical limitations were encountered and documented, such as unexpected tone differences in PWM setup, interrupt latency instability in TinyGo, and subtle garbage collection-induced delays. These issues informed realistic assessments of language suitability.

Objective 4: Use disassembly insights to contextualise performance Disassembly of compiled `.elf` binaries revealed function call depth, stack frame size, and register usage patterns. These insights explained observed performance differences, particularly in recursive calls and peripheral setup routines. Comparisons of function prologues and epilogues were used to relate latency back to instruction-level design.

Objective 5: Assess suitability for real-time constraints Interrupt latency benchmarks revealed significant differences in real-time responsiveness. C offered more deterministic low-latency behaviour, while TinyGo's runtime overhead led to greater variability. These results were critical in assessing each language's viability for real-time embedded systems.

Objective 6: Provide developer guidance for language selection A series of practical recommendations were proposed based on empirical results. C is better suited for performance-critical tasks, while TinyGo remains compelling for rapid prototyping or resource-constrained deployments. Hybrid architectures were suggested as a way to combine strengths of both ecosystems.

Summary Table: Objectives vs Outcomes

Objective	Outcome Achieved
Benchmark software and hardware tasks	Full benchmark suite executed with consistent methodology and logging
Quantify binary size, RAM, jitter	Measured across all tasks; results presented with statistical variation
Identify implementation challenges	Interrupt jitter, PWM mismatches, GC latency all documented
Use disassembly insights	.S and .map files analysed to explain performance deltas
Assess real-time suitability	C shown to outperform TinyGo for latency-sensitive operations
Provide developer guidance	Practical trade-offs and hybrid design suggestions presented

Table 5.1: Project objectives and corresponding outcomes

In summary, all original project aims were successfully addressed, with several areas exceeding expectations. Benchmarking was comprehensive, disassembly insights proved essential, and practical engineering challenges were critically analysed. The resulting recommendations offer tangible value to embedded developers evaluating TinyGo for real-world deployment.

5.6.5 Personal Outcomes

This project also represented a significant learning opportunity, including:

- Gaining practical experience with the RP2040 microcontroller.
- Brushing up on C programming for performance optimisation.
- Learning TinyGo and its nuances for the first time.
- Developing formal benchmarking skills, including data analysis and disassembly interpretation.
- Building a deeper understanding of real-time systems and embedded software trade-offs.

Chapter 6

Legal, Social, Ethical and Professional Issues

6.1 Statement of Ethics

This project was carried out fully aware of the legal, social, ethical, and professional (LSEP) obligations expected of computing professionals. A Self-Assessment for Governance and Ethics (SAGE) form was completed, which confirmed that formal ethical approval was not required, as the project did not involve human participants, personal data, or sensitive materials.

6.2 Compliance with the BCS Code of Conduct

As a student member of the British Computer Society (BCS), this work adheres to the principles of the BCS Code of Conduct, specifically:

- **Public Interest:** The benchmarking outcomes are intended to promote safer and more predictable embedded software development, benefiting society by guiding language choice in critical applications such as healthcare, automation, and IoT.
- **Professional Competence and Integrity:** All technical claims are substantiated through reproducible experiments and clearly reported metrics. The project scope, limitations, and methodology are transparently documented.
- **Duty to Relevant Authority:** The project was conducted under the academic over-

sight of the University of Surrey per its policies on research ethics, software conduct, and intellectual property.

- **Enhancement of IT:** By exploring the implications of abstraction, garbage collection, and toolchain maturity in embedded contexts, the project contributes to ongoing efforts to improve software reliability and resilience in constrained environments.

6.3 Data Protection and Confidentiality

No personal or confidential data was handled. All benchmark data was synthetically generated on microcontroller hardware and pertains exclusively to technical metrics such as execution time and memory usage. These data fall outside the scope of the UK General Data Protection Regulation (GDPR) and the Data Protection Act (2018). However, good research practice was observed in the secure storage and archival of results.

6.4 Intellectual Property and Licensing

All source code developed during the project is original work unless explicitly cited. Third-party libraries and SDKs (e.g., Pico SDK and TinyGo runtime) are used following their respective open-source licences. I, the author, retain the intellectual property rights to the analytical content and associated benchmarks presented in this dissertation under the University of Surrey's Code of Practice on Intellectual Property and Exploitation.

6.5 Social Responsibility and Broader Impact

This work supports responsible embedded systems development by highlighting the trade-offs between traditional low-level languages and modern memory-safe alternatives. The findings contribute to informed decision-making in contexts where timing precision and software reliability are critical. The project indirectly supports the aims of:

- **SDG 9: Industry, Innovation and Infrastructure** — through improved tooling insights for embedded engineers.

- **SDG 11: Sustainable Cities and Communities** — by promoting safe and efficient firmware for IoT and urban technology platforms.
- **SDG 3: Good Health and Well-being** — via relevance to safety-critical embedded systems in medical devices.

6.6 Academic Integrity

The work complies fully with the University of Surrey's academic integrity policies. All references, including codebases, technical datasheets, and academic literature, are correctly cited. No plagiarism occurred, and the structure and conclusions of this dissertation are entirely my own.

.1 Benchmark Data Tables

.1.1 Fibonacci

Table 1: Execution Time and Standard Deviation (µs) — Fibonacci (Iterative and Recursive)

Variant	**C**		**TinyGo**	
	Mean	**SD**	**Mean**	**SD**
Recursive (n=10)	96.0	5.52	22.0	2.83
Recursive (n=20)	2658.6	5.41	1545.8	4.76
Recursive (n=30)	296328.0	14.11	188488.0	0.71
Recursive (n=35)	3287688.2	4.02	2090262.0	1.10
Iterative (n=10)	4.0	0.0	24.8	0.45
Iterative (n=20)	1.6	0.55	5.8	1.64
Iterative (n=30)	2.0	0.0	5.2	0.45
Iterative (n=35)	2.0	0.0	5.6	0.55

Table 2: Binary Size and RAM Usage — Fibonacci

Benchmark	**C Binary Size (KB)**	**TinyGo Binary Size (KB)**	**C RAM (bytes)**	**TinyGo RAM (bytes)**
Fibonacci (both variants)	33.2	9.35	11932	3280

.1.2 Bubble Sort

Table 3: Execution Time and Standard Deviation (µs) — Bubble Sort

Array Size	**C**		**TinyGo**	
	Mean	**SD**	**Mean**	**SD**
10 elements	16.4	11.8	32.6	0.5
50 elements	168.2	8.4	160.0	3.4
100 elements	607.4	8.8	614.6	2.3

Table 4: Binary Size and RAM Usage — Bubble Sort

Benchmark	**C Binary Size (KB)**	**TinyGo Binary Size (KB)**	**C RAM (bytes)**	**TinyGo RAM (bytes)**
Bubble Sort	32.4	9.09	11932	3280

.1.3 Quick Sort

Table 5: Execution Time and Standard Deviation (µs) — Quick Sort

Array Size	**C**		**TinyGo**	
	Mean	**SD**	**Mean**	**SD**
10 elements	25.4	10.3	41.2	0.4
50 elements	169.2	11.4	184.2	2.9
100 elements	613.4	3.8	686.0	4.4

Table 6: Binary Size and RAM Usage — Quick Sort

Benchmark	**C Binary Size (KB)**	**TinyGo Binary Size (KB)**	**C RAM (bytes)**	**TinyGo RAM (bytes)**
Quick Sort	32.45	9.15	11932	3280

.1.4 Loop Overhead

Table 7: Execution Time and Standard Deviation (µs) — Loop Overhead

Loop Iterations	**C**		**TinyGo**	
	Mean	**SD**	**Mean**	**SD**
1K	92.6	15.97	63.6	2.73
10K	778.0	13.43	414.2	3.54
100K	7240.8	3.19	4011.6	0.49
1M	72046.8	1.47	40011.8	0.40

Table 8: Binary Size and RAM Usage — Loop Overhead

Benchmark	**C Binary Size (KB)**	**TinyGo Binary Size (KB)**	**C RAM (bytes)**	**TinyGo RAM (bytes)**
Loop Overhead	32.34	9.04	11932	3280

.1.5 Matrix Multiplication

Table 9: Execution Time and Standard Deviation (µs) — Matrix Multiplication

Matrix Size	**C**		**TinyGo**	
	Mean	**SD**	**Mean**	**SD**
10x10	128.8	8.38	285.4	3.77
20x20	729	1.67	1914.2	0.4

Table 10: Binary Size and RAM Usage — Matrix Multiplication

Benchmark	**C Binary Size (KB)**	**TinyGo Binary Size (KB)**	**C RAM (bytes)**	**TinyGo RAM (bytes)**
Matrix Multiplication	32.55	9.34	11932	3280

.1.6 FFT (Radix-2)

Table 11: Execution Time and Standard Deviation (µs) — FFT (Radix-2)

FFT Task	**C**		**TinyGo**	
	Mean	**SD**	**Mean**	**SD**
128-point FFT	4946.6	2.245	5262.6	0.49

Table 12: Binary Size and RAM Usage — FFT (Radix-2)

Benchmark	**C Binary Size (KB)**	**TinyGo Binary Size (KB)**	**C RAM (bytes)**	**TinyGo RAM (bytes)**
128-point FFT	33.5	16.88	11932	3280

.1.7 ADC Read

Table 13: Execution Time and Standard Deviation (µs) — ADC Read (Average of 1000 reads)

Task	**C**		**TinyGo**	
	Mean	**SD**	**Mean**	**SD**
Read ADC (GPIO26)	2	0	6	0

Table 14: Binary Size and RAM Usage — ADC Read

Benchmark	**C Binary Size (KB)**	**TinyGo Binary Size (KB)**	**C RAM (bytes)**	**TinyGo RAM (bytes)**
ADC Read	32.7	9.34	11932	3288

.1.8 GPIO Toggle

Table 15: Execution Time and Standard Deviation (µs) — GPIO Toggle (1000 toggles)

Task	**C**		**TinyGo**	
	Mean	**SD**	**Mean**	**SD**
1000 toggles on GPIO2	50.4	0.49	63.4	2.33

Table 16: Binary Size and RAM Usage — GPIO Toggle

Benchmark	**C Binary Size (KB)**	**TinyGo Binary Size (KB)**	**C RAM (bytes)**	**TinyGo RAM (bytes)**
GPIO Toggle	32.7	13.63	11932	3280

.1.9 PWM Setup

Table 17: Execution Time and Standard Deviation (µs) — PWM Setup

Task	**C**		**TinyGo**	
	Mean	**SD**	**Mean**	**SD**
Configure and start PWM on GPIO15	8.4	0.49	43.6	0.49

Table 18: Binary Size and RAM Usage — PWM Setup

Benchmark	**C Binary Size (KB)**	**TinyGo Binary Size (KB)**	**C RAM (bytes)**	**TinyGo RAM (bytes)**
PWM Setup	32.44	8.94	11932	3280

.1.10 Interrupt Latency

Table 19: Latency and Standard Deviation (µs) — GPIO Interrupt Latency

Event	**C**		**TinyGo**	
	Mean	**SD**	**Mean**	**SD**
GPIO14 Rising Edge to ISR (GPIO15 High)	96.5	82.5	482.9	281.2

Table 20: Binary Size and RAM Usage — Interrupt Latency

Benchmark	**C Binary Size (KB)**	**TinyGo Binary Size (KB)**	**C RAM (bytes)**	**TinyGo RAM (bytes)**
Interrupt Handling (single press latency)	34.00	9.13	11956	3768

.1.11 UART Transmission

Table 21: Execution Time and Standard Deviation (µs) — UART TX (per message)

Message Sent	**C**		**TinyGo**	
	Mean	**SD**	**Mean**	**SD**
"Hello from main Pico" (×100)	1794.132	0.004	1847.502	0.023

Table 22: Binary Size and RAM Usage — UART TX

Benchmark	**C Binary Size (KB)**	**TinyGo Binary Size (KB)**	**C RAM (bytes)**	**TinyGo RAM (bytes)**
UART TX (100× full messages)	33.13	13.91	11936	3408

.1.12 I2C Write

Table 23: Execution Time and Standard Deviation (µs) — I2C Write

Message Type	**C**		**TinyGo**	
	Mean	**SD**	**Mean**	**SD**
Write 4-byte message to 0x42 (×100)	121.23	0.06	121.152	0.004

Table 24: Binary Size and RAM Usage — I2C Write

Benchmark	**C Binary Size (KB)**	**TinyGo Binary Size (KB)**	**C RAM (bytes)**	**TinyGo RAM (bytes)**
I2C Write (100× messages)	33.3	14.22	11940	3280

Bibliography

AG, C. (2016), 'Brake systems of the future'. Accessed March 2025.
URL: https://www.continental.com/en/press/studies-publications/technology-dossiers/brake-systems-of-the-future/

Cimpanu, C. (2019), 'Microsoft: 70 percent of all security bugs are memory safety issues'. Accessed March 2025.
URL: https://www.zdnet.com/article/microsoft-70-percent-of-all-security-bugs-are-memory-

El-Hadedy, M., Hua, R., Saqib, S., Yoshii, K., Hwu, W.-M. & Margala, M. (2023), Bltesti: Benchmarking lightweight tinyjambu on embedded systems for trusted iot, *in* '2023 IEEE 36th International System-on-Chip Conference (SOCC)', pp. 1–6. Accessed March 2025.

Elliott, C., Finne, S. & de Moor, O. (2003), 'Compiling embedded languages', *Journal of Functional Programming* **13**(3), 455–481. Accessed March 2025.
URL: https://www.cambridge.org/core/journals/journal-of-functional-programming/article/compiling-embedded-languages/4B0A7526CC16907F445CCF27277E9B9B

Gay, D., Levis, P. & Culler, D. (2006), Memory safety and untrusted extensions for tinyos, Technical report, University of Utah. Accessed March 2025.
URL: https://cs.utah.edu/docs/memory-safety-tinyos.pdf

Mastery, T. (2023), 'Tinygo: A comprehensive guide', https://dotcommagazine.com/2023/12/tinygo-a-comprehensive-guide/. Accessed March 2025.

Plauska, I., Liutkevičius, A. & Janavičiūtė, A. (2023), 'Performance evaluation of c/c++, micropython, rust and tinygo programming languages on esp32 microcontroller', *Electronics* **12**(1), 143. Accessed March 2025.
URL: https://www.mdpi.com/2079-9292/12/1/143

Powers, C. R. (1998), A review of performance analysis (benchmarking) approaches for embedded microprocessors and microcontrollers, Master's thesis, University of Texas at Austin. Accessed March 2025.
URL: https://users.ece.utexas.edu/~bevans/students/ms/chuck_powers/ms.pdf

Raspberry Pi Foundation (2023), 'Getting started with raspberry pi pico', https://datasheets.raspberrypi.com/pico/getting-started-with-pico.pdf. Accessed April 2025.

Raspberry Pi Ltd. (2021), *Raspberry Pi Pico Datasheet*. Accessed April 2025.
URL: https://datasheets.raspberrypi.com/pico/pico-datasheet.pdf

Raspberry Pi Ltd. (2023), *Raspberry Pi Pico C SDK*. Accessed April 2025.
URL: https://datasheets.raspberrypi.com/pico/raspberry-pi-pico-c-sdk.pdf

Reddit Users (2023), 'Is tinygo real-time capable?'. Accessed March 2025.
URL: https://www.reddit.com/r/golang/comments/13rkvau/is_tinygo_realtime_capable/

Samefors, A. & Sundman, F. (2023), Investigating energy consumption and responsiveness of low power modes in micropython for stm32wb55, Master's thesis, Jönköping University, School of Engineering. Accessed March 2025.
URL: https://www.diva-portal.org/smash/record.jsf?pid=diva2:1778426

Sujeeth, A. K., Lee, H., Brown, K. J., Rompf, T., Chafi, H., Wu, M., Nowak, K. R. J., Odersky, M. & Olukotun, K. (2014), Delite: A compiler framework for parallel embedded dsls, *in* 'Proceedings of the 2014 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)', pp. 113–124. Accessed March 2025.
URL: https://dl.acm.org/doi/abs/10.1145/2584665

TinyGo Authors (2024), 'Tinygo: A go compiler for small places'. Accessed April 2025.
URL: https://tinygo.org

TinyGo Authors (2025*a*), 'Low power'. Accessed April 2025.
URL: https://tinygo.org/docs/guides/low-power/

TinyGo Authors (2025*b*), 'Optimizing binaries'. Accessed April 2025.
URL: https://tinygo.org/docs/guides/optimizing-binaries/

TinyGo Authors (2025*c*), 'Tips, tricks and gotchas'. Accessed April 2025.

URL: `https://tinygo.org/docs/guides/tips-and-tricks/`