Article

Framework Design for the Dynamic Reconfiguration of IoT-Enabled Embedded Systems and "On-the-Fly" Code Execution

Elmin Marevac [1], Esad Kadušić [2], Nataša Živić [3,*], Nevzudin Buzađija [1] and Samir Lemeš [1]

[1] Polytechnic Faculty, University of Zenica, 72000 Zenica, Bosnia and Herzegovina; elmin.marevac@unze.ba (E.M.); nevzudin.buzadjija@unze.ba (N.B.); samir.lemes@unze.ba (S.L.)
[2] Faculty of Educational Sciences, University of Sarajevo, 71000 Sarajevo, Bosnia and Herzegovina; ekadusic@pf.unsa.ba
[3] Faculty of Digital Transformation, Leipzig University of Applied Sciences, 04277 Leipzig, Germany
* Correspondence: natasa.zivic@htwk-leipzig.de

Abstract: Embedded systems, particularly when integrated into the Internet of Things (IoT) landscape, are critical for projects requiring robust, energy-efficient interfaces to collect real-time data from the environment. As these systems become complex, the need for dynamic reconfiguration, improved availability, and stability becomes increasingly important. This paper presents the design of a framework architecture that supports dynamic reconfiguration and "on-the-fly" code execution in IoT-enabled embedded systems, including a virtual machine capable of hot reloads, ensuring system availability even during configuration updates. A "hardware-in-the-loop" workflow manages communication between the embedded components, while low-level coding constraints are accessible through an additional abstraction layer, with examples such as MicroPython or Lua. The study results demonstrate the VM's ability to handle serialization and deserialization with minimal impact on system performance, even under high workloads, with serialization having a median time of 160 microseconds and deserialization having a median of 964 microseconds. Both processes were fast and resource-efficient under normal conditions, supporting real-time updates with occasional outliers, suggesting room for optimization and also highlighting the advantages of VM-based firmware update methods, which outperform traditional approaches like Serial and OTA (Over-the-Air, the ability to update or configure firmware, software, or devices via wireless connection) updates by achieving lower latency and greater consistency. With these promising results, however, challenges like occasional deserialization time outliers and the need for optimization in memory management and network protocols remain for future work. This study also provides a comparative analysis of currently available commercial solutions, highlighting their strengths and weaknesses.

Keywords: IoT; embedded systems; dynamic reconfiguration; hot reloads; virtual machine; hardware-in-the-loop

check for updates

Academic Editors: Manuel José Cabral dos Santos Reis and Carlos Serôdio

Received: 8 November 2024
Revised: 23 December 2024
Accepted: 2 January 2025
Published: 7 January 2025

Citation: Marevac, E.; Kadušić, E.; Živić, N.; Buzađija, N.; Lemeš, S. Framework Design for the Dynamic Reconfiguration of IoT-Enabled Embedded Systems and "On-the-Fly" Code Execution. *Future Internet* **2025**, *17*, 23. https://doi.org/10.3390/fi17010023

1. Introduction

1.1. Embedded Systems and Dynamic Configuration Challenges

The rapid advancement of technology has led to an intensive increase in the presence of embedded systems across various domains, including industrial automation, automotive systems, and consumer electronics. These systems are designed to perform specific tasks efficiently and reliably, often with minimal human intervention [1]. However, the growing complexity and dynamic nature of modern applications have created the need for more flexible and adaptable embedded systems. Traditional embedded systems, typically designed to run a single, fixed set of instructions, are no longer sufficient for contemporary applications requiring frequent updates, real-time adjustments, and dynamic reconfiguration. One of the key challenges in developing embedded systems is the need for efficient and flexible code execution. Embedded systems often operate in resource-constrained environments where memory and processing power are limited. These constraints demand optimized code and efficient resource management. However, the traditional approach of compiling code into machine-specific instructions can be limiting, as it does not allow easy updates or modifications [2].

To address these limitations, researchers and developers have explored the use of virtual machines (VMs) and interpreters in embedded systems. Virtual machines provide an abstraction layer between hardware and software, enabling the execution of code that is not specific to the underlying hardware [3]. This abstraction facilitates the creation and use of a unified and often platform-independent design, simplifying portability and maintenance. Interpreters, on the other hand, execute code line-by-line, allowing dynamic code changes without the need for recompilation. This capability is especially useful where frequent updates or modifications are needed [2]. By using interpreters, developers can make code changes "on the fly" without requiring a complete system restart. Integrating virtual machines and interpreters offers several advantages:

- Dynamic system reconfiguration: This allows changes in system behaviour without physical reconfiguration, which is particularly useful in applications with frequently changing requirements, such as industrial automation or autonomous vehicles.
- Platform independence and enhanced portability: Virtual machines and interpreters enable the execution of code that is not tied to specific hardware, which is critical in embedded systems where hardware can change or evolve over time. By using a virtual machine, developers can ensure that their code remains compatible with various hardware platforms, reducing the need for extensive rewriting or recompilation. The interpreter within the virtual machine can further abstract the code, making it easier to adapt to various environments such as sandboxing untrusted applications or running potentially harmful scripts in a controlled environment.
- Simplified debugging and testing: The combination of virtual machines and interpreters allows developers to test code changes immediately, as interpreters execute code in real time. This setup is useful for debugging and development, as it enables developers to identify and resolve issues on the fly without needing a full system rebuild.
- Resource efficiency: By combining virtual machines with interpreters, it is possible to execute lightweight, isolated processes that do not require the full overhead of recompiling or redeploying applications. This setup is particularly useful in environments where computing resources are limited, such as IoT devices or embedded systems.
- Improved cross-language support: Virtual machines can support multiple programming languages within the same environment, and interpreters can enable interoperability between these languages. For instance, a Python interpreter running on a Java virtual machine can allow for cross-language calls, enhancing the flexibility of multi-language applications.
- Increased security and reliability: Executing code in a virtualized environment enables developers to isolate the system from potential security threats and ensure the system remains stable and reliable, even in the event of unexpected changes [4].

The development of microcontrollers faces ongoing challenges concerning efficiency, flexibility, and adaptability. The traditional approach to programming microcontrollers, which relies on compiling code into machine instructions, has proven to be limiting and inflexible in aspects such as code maintenance and incremental development. Compiled code is bound to a specific hardware platform, making updates, modifications, and code portability to other microcontrollers difficult. Additionally, this type of program code is inherently static and does not allow for dynamic changes in program logic. It is also essential to consider low-level programming, which is unavoidable due to limited memory and processing resources and requires specialized knowledge, skills, and insight into the internal system structure. Some standard debugging tools are also limited in this regard [5].

In recent years, this has led to the increased popularity of interpreted languages such as MicroPython and Lua in this area. These languages offer flexibility and portability, enabling code execution across different hardware platforms without the need for recompilation. Moreover, the simplicity and higher-level nature of these languages allow for faster iteration and development, attracting a broader circle of developers. Their interpreted nature allows for dynamic reconfiguration and rapid changes in system behaviour, which is essential for modern embedded applications. Their small size, efficiency, and low resource requirements make them suitable for operation in microcontrollers. This approach holds great potential for the near future, where growing demands for flexibility and the acceptance of embedded system integration with existing platforms will become critical factors for software development and maintenance [6]. This will also likely lead to the expansion of developer communities and repositories of libraries and tools, which, although currently modest with some basic native method calls, may make this approach more attractive and catalyse advancements in hardware with greater processing power and memory to support such platforms [7].

As the demands for intelligent, connected, and adaptable embedded systems continue to grow, the next natural step would be the creation of frameworks for dynamic configurations and procedure execution to facilitate the development of innovative projects related to IoT, robotics, the automotive industry, and other industries subject to frequent adaptations with minimized delays. In this case, the mentioned concepts and languages would form the foundation of such frameworks, providing time-extended platform independence and contributing to both horizontal and vertical scalability [8]. Additional integrated development tools would offer a comprehensive environment for efficient code creation, testing, and deployment, with simulator support for project verification [4].

The aim of this paper is to present the design of a framework with support for hot reloads, code, and data persistence incorporated with a native interface for binding external libraries suitable for operation in microcontrollers with all hardware limitations by using an existing virtual machine. This project will serve as the basis for three additional components: a microcontroller-side interpreter that efficiently executes the created intermediate code and modifies it on demand, a client-side integrated development environment for creating, compiling, and debugging code with simulator-based verification, and a compiler that creates intermediate code in the established format, suitable for transfer to microcontrollers registered on the local network. Although a performance loss in such circumstances is expected and inevitable, the idea of such an environment should primarily allow development teams to flexibly compromise between improved system adaptability to behaviour modification requirements and sensor handling and resource savings for a more efficient processing of program code, incoming data, and measurements.

1.2. Related Studies

In recent years, a significant number of notable studies have presented various techniques for improving the dynamic behaviour of embedded systems at the data storage level, as well as for modifying and executing program logic on demand, yielding notable results. The study entitled "Runtime Environment for Dynamically Reconfigurable Embedded

Systems" [9] explores the development of a runtime environment specifically designed to support dynamic reconfiguration in embedded systems, providing an identical execution context for both software and hardware, as well as the capability to restructure the process structure implemented at both levels. This study is significant as it establishes the primary criteria required for the stability and platform independence of such projects, the approach to synthesizing the interface for integrating high-level modules, and the concept of a reconfiguration manager. The methodologies presented in this study were crucial for designing native high-level modules within the integrated development environment (IDE) and for their final implementation and data transfer to microcontrollers, with the manager acting as a mediator in the exchange of requests and responses.

For insight into the process of hardware reconfiguration using bitstreams and standard FPGA circuits combined with an external controller, the study "Dynamic Reconfigurability in Embedded System Design" [10] has been deemed useful. The primary focus of this research is on the implementation of a three-layer architecture for data transfer from a high-level client system, partial bitstream validation for each functionality, and the generation of a new structure for low-level reprogramming. This approach is also applicable to virtual machines at a higher abstraction layer, with a gradual transition from source code to intermediate code and, ultimately, to calling system procedures on the microcontroller to implement changes with prior validation. Additionally, the FPGA circuits applied in this study could be used as a potential platform extension, especially for real-time image and signal processing algorithms.

Another study titled "A Dynamic Reconfiguration Scheme for Embedded System Based on Multi-core DSP" [11] introduces an innovative approach to dynamic reconfiguration tailored specifically for multi-core Digital Signal Processors (DSPs), which are critical for high-performance embedded systems such as those used in aviation. By designating one core as a management core, the scheme enables the efficient time multiplexing of multiple algorithms on a single core, maximizing hardware resource utilization without interrupting system operations. Additionally, the use of a local reset mechanism allows seamless software switching, ensuring real-time operation with a reconfiguration time of just 4 milliseconds for a 128 KB program. The contribution of this study is particularly evident in the firmware design for a microcontroller that simultaneously processes user requests and virtual machine instructions in a distributed manner, while local reset only applies to initiating a new instance of the program with the received options and code.

In the paper "A Low-Overhead Script Language for Tiny Networked Embedded Systems" [12], a new scripting language named "SCript" is presented, specifically designed for resource-constrained sensor networks. The authors developed an interpreter for this language that runs on the popular MSP430 microcontroller and benchmarked its performance, showing comparable execution times to existing virtual machines while maintaining a similar memory footprint. SCript provides features such as conditional branching, loops, functions, and scoped variables, making it more powerful than simpler scripting approaches. These characteristics make SCript a flexible and efficient means for dynamically reprogramming sensor networks without requiring physical hardware modifications, reducing development time and enabling easier adaptation to changing environmental conditions. Most of the best practices and approaches described in this study have been applied to this project.

The paper "Experience with Safe Dynamic Reconfigurations in Component-Based Embedded Systems" [6] discusses ensuring dynamic reconfiguration based on secure and efficient updates without service interruption. The authors propose a component-based framework and a domain-specific language (DSL), FScript, for effectively managing reconfigurations. Their approach, implemented on the "Cognichip" platform, offers solutions to achieve flexibility, security, and simplicity in reconfigurations, applicable to interpreted languages. In this context, functional units of the execution environment can be treated as separate components, with the virtual machine instance remaining active during microcontroller operation.

1.3. Language Interpreters and Virtual Machines in Embedded Systems

One of the concepts of used language interpreters for dynamic code execution is that they usually generate a set of machine code-like instructions (often called "bytecode") instead of directly generating machine code for the targeted platform, which not only allows programmers to follow program execution instantly, but also to develop and deploy applications more efficiently, often in combination with agile practices. Even though these execution environments do not share common grammar, architecture, or program structures, most of them share some common examples, approaches, best practices, and algorithms for sufficient or acceptable portability and efficiency, as described in the research paper titled "Design and Implementation of a Virtual Machine for a Dynamic Object-Oriented Programming Language", which also describes Stella: a functional, procedural, class- and prototype-based object-oriented and modular general-purpose programming language developed to provide a flexible and scalable programming environment supporting both low- and high-level procedures, allowing for extension within other languages such as C, C++, or Java. Most of the virtual machine's existing concepts remain unchanged, though some adaptations to the native interface, bytecode serialization, and library integration for C and C++ projects were necessary to improve system efficiency, effectiveness, and compatibility [7].

Virtual machines (VMs) have become a key component of modern IT infrastructures, enabling efficient resource utilization, flexibility, and application isolation. However, their application to resource-constrained devices, such as embedded systems and microcontrollers, presents a significant bottleneck that requires careful consideration, with one primary challenge being the inherent limitations of processing and memory resources [13]. These devices are designed to be compact, energy-efficient, and cost-effective, often sacrificing processing power and memory capacity to achieve these features, which can seriously limit their ability to support virtual machines effectively. Implementing VMs as interpreters for microcontrollers involves addressing several key limitations:

- Resource Constraints—This is usually the most emphasized and noticeable issue, as microcontrollers typically have limited processing power and memory capacity. Designed to be compact and energy-efficient, these devices often lack the RAM necessary for running complex applications or VMs efficiently. For example, a typical microcontroller averages 1 MB of RAM, which can be quickly exhausted by adding numerous libraries for communication protocols and data processing; a VM as an abstraction layer further increases the consumption of processing resources, leading to system instability and overheating.
- Energy Consumption—Many embedded systems operate on batteries, making energy efficiency a crucial factor in their design and operation. The virtualization process can lead to increased energy consumption due to additional CPU and memory load. Studies show that virtualization increases energy consumption compared to traditional approaches, which can be particularly problematic for devices where extended battery life without frequent charging is essential [13,14].
- Performance Issues—The additional load can also lead to increased latency and reduced responsiveness. Many systems have stringent real-time processing requirements, where delays can compromise the functionality of critical applications. Thus,

managing the performance degradation caused by virtualization is necessary to ensure that systems meet their operational requirements.

- Complexity and Maintenance Overhead: Implementing a VM layer in a microcontroller introduces additional complexity, both in terms of development and maintenance. Developers need to manage the virtualized environment alongside the microcontroller's hardware constraints, which can complicate debugging, increase the potential for errors, and require more specialized knowledge.
- Limited I/O and Peripheral Access: Microcontrollers are often directly connected to specific hardware peripherals (e.g., sensors, actuators) that need low-latency, direct access. A VM can obscure or restrict direct access to these peripherals, adding complexity when configuring device drivers or accessing hardware-specific features.
- Scalability Constraints—Increasing the number of required libraries and functional modules in the system intensifies the physical device load. Given the aforementioned limited resources, vertical and horizontal scaling possibilities must be considered, requiring adjustments to the VM structure as well. In the infrastructure of networked embedded systems, vertical scaling involves extending an existing node with more resources, such as optimal memory configurations as needed or hardware upgrades with Flash or external EEPROM chips, while horizontal scaling involves adding new nodes interconnected with the rest via appropriate communication protocols. Regardless of the chosen approach, the basic VM specification should include essential procedures for memory access and establishing communication through the selected protocol [11].

1.4. Research Objectives

Therefore, the primary objective of this research is to develop and implement a framework that enables efficient dynamic reconfiguration and on-demand code execution in embedded systems, thereby enhancing their functionality and adaptability. This goal includes the exploration of existing solutions, the development of new methodologies and tools, and the evaluation of the framework's efficiency in practical applications. The specific objectives of this research, based on identified challenges and research topics, include the following:

- Developing a comprehensive framework that supports the dynamic reconfiguration of embedded systems, covering all aspects necessary for the effective management of reconfiguration and code execution;
- Investigating and analysing existing technologies and tools used for dynamic reconfiguration, identifying their strengths and limitations, with the goal of developing an innovative solution that overcomes these challenges;
- Creating an integrated development environment that enables programmers to efficiently create, test, and deploy applications for embedded systems, including tools for simulation, debugging, and optimization;
- Testing the efficiency of the developed framework and development environment through practical examples in the form of unit tests to assess the performance, flexibility, and reliability of the system under real-world conditions;
- Developing guidelines, recommendations, and suggestions for implementing dynamic reconfiguration in industrial applications, assisting organizations in successfully integrating new technologies into their existing systems;
- Contributing to the academic community through a review and analysis of the research results, fostering further discussion and research in this field.

1.5. The Study Roadmap

This paper is organized in five sections. The introduction to this research, as well as challenges and study objectives were provided in this section, Section 1. Section 2 discusses the state of the art, focusing on existing frameworks and technologies for dynamic reconfiguration. Section 3 is the core of this research, outlining the materials and methods used to develop and implement the proposed system framework. In Section 4, the results of this study, including stress tests and comparative analysis, are presented. In Section 5, the study conclusions and future directions for enhancing the proposed framework are discussed.

2. State of the Art

In this section, the authors discuss the state of the art of existing methodologies and frameworks enabling dynamic reconfiguration, as well as examples of existing framework implementations as preparation for approaching the development of the proposal of this paper.

With the gradual shift towards agile methodologies and processes for Continuous Integration and Continuous Deployment (referred to as CI/CD) in project workflows for embedded systems, there has been an increasing demand for dynamic environments that facilitate the efficient and straightforward execution of test scripts, dynamic code evaluation, and results analysis. Developing correct and efficient programs for microcontrollers is highly challenging and time-consuming. Microcontrollers are typically programmed in low-level languages such as C, which makes debugging and maintenance difficult [14]. Furthermore, their constraints and minimalism often negatively impact the development lifecycle, hindering development teams from fully adopting an agile approach for several key reasons:

- Agile methodologies advocate for a flexible approach with rapid responses to client requirements. This is not easily achievable by directly modifying the firmware of devices, which would instead require updates or system recompilation, introducing additional delays that not only slow down system response but also hinder the critical testing and verification process necessary for continuous integration. Additionally, the development cycle is inherently long and slow, as even minor changes require a complete re-run of the entire outlined process [10].
- System debugging limitations pose additional challenges. Most debugging tools cover only the most popular platforms and can impact the reliability and objectivity of performance testing results. In such situations, development teams often rely on logging messages and microcontroller states with data on resource usage, though this approach still fails to provide detailed insights into the system's status and operation [4].
- Low-level microcontroller setup, where component initialization errors and memory management issues are frequent, can lead to system crashes without clear error information. Error data and interrupt signals are commonly presented in the form of stack and memory dumps, which further slow system verification down due the need to reproduce the error, analyse feedback data, and finally identify the cause. These drawbacks not only hinder production software recovery in response to user feedback, but also disrupt the entire CI/CD pipeline for complex systems [14].

Some of these obstacles can be mitigated by using one of the high-level programming languages available for microcontroller programming. Their dynamic and often interpreted nature provides flexible, incremental changes in support of virtual platforms and CI/CD, offering a range of auxiliary tools and APIs for remote debugging, system monitoring, and instance state management. However, while these languages offer greater abstraction for microcontroller programming, they also introduce a new set of issues: debugging support often relies on using "print" statements, the language runtime consumes substantial on-chip space regardless of functionality used, and they are considerably slower than low-level languages, making driver development for devices inefficient [4]. Nevertheless, embedded system development is a matter of compromise: high-level languages simplify development and maintenance, enabling faster iterations and reducing complexity, though at the cost of reduced performance and increased memory demands. The world of microcontrollers and programming languages for their development has undergone significant growth and application. A wide range of programming languages have been adapted for various hardware platforms, such as Forth, BASIC, Java, Python, Lua, and more [2]. The following sections describe some of the popular approaches for programming the ESP family of microcontrollers, the platform on which the created framework was primarily tested.

2.1. Zerynth Virtual Machine

The Zerynth Virtual Machine [15] is a powerful platform designed to execute Python 3.4 scripts on embedded systems, offering a high level of code reusability across different boards. It supports various high-level Python features, including modules, classes, multi-tasking, callbacks, timers, and exceptions, as well as hardware-related functions such as interrupts, PWM, and digital I/O. The VM is built on top of real-time operating systems (RTOS), such as CHIBIOS or FreeRTOS, utilizing an abstraction layer called VOSAL (Virtual Machine Operating System Abstraction Layer). This architecture allows for easy porting to different RTOS platforms and comprises several key components [15].

The core functionality of this platform, as presented in Figure 1, is provided by its bytecode interpreter, which executes Python bytecode that the Zerynth compiler generates from Python scripts [16]. These bytecode objects include instructions and metadata necessary for managing memory and handling errors, with each opcode represented by one byte and optional arguments of up to four bytes. The interpreter processes these instructions sequentially within the active thread until a stop instruction is encountered, offering a design that closely resembles Python but with opcodes tailored for embedded systems. To ensure thread safety during opcode execution, the Global Interpreter Lock (GIL) allows only one Python thread to execute bytecode at any moment, releasing control to other threads when a thread either sleeps or its quantum time expires, thus preventing race conditions.

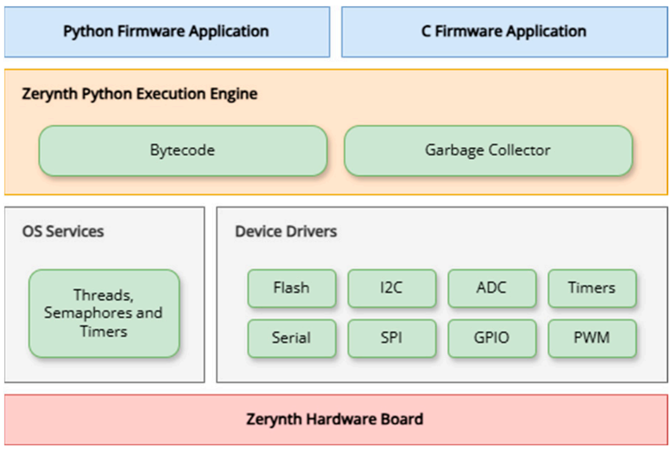


Figure 1. Zerynth virtual machine architecture block diagram (adapted from [15]).

Memory management within the VM is handled by the garbage collector (GC), which periodically scans for unused objects and removes them through a mark-and-sweep algorithm, thereby optimizing memory usage and simplifying lifecycle management. The interrupt thread within Zerynth is a high-priority thread that executes bytecode triggered by hardware interrupts, allowing for typical memory allocation but requiring minimal processing time to prevent delays in handling other interrupts. The Virtual Machine Operating System Abstraction Layer (VOSAL) provides essential abstraction for interfacing with the RTOS, including functions for multitasking structures like threads and semaphores, supporting the hybrid execution of C and Python code and enhancing portability across RTOS platforms. Additionally, the Virtual Machine Hardware Abstraction Layer (VHAL) acts as an interface for controlling microcontroller peripherals, such as serial ports, SPI, I2C, ADC, and PWM, with specific implementations for each microcontroller family, ensuring a consistent hardware API regardless of the underlying hardware [15].

To optimize the virtual machine for embedded systems, certain Python features have been modified or removed. For instance, integer values are implemented as 32-bit signed numbers, object sizes are reduced, and lists and dictionaries are limited to 65,536 elements. Compilation is moved out of the language, and features like closures, generators, and decorators are either removed or added modularly. True multithreading with priorities has been introduced, with each Zerynth thread functioning as a native thread of the real-time operating system [15].

2.2. MicroPython

Python [17] is often regarded as a high-level programming language offering significant abstraction, yet it also proves to be an excellent tool in hardware development. MicroPython, an open-source, scalable port of CPython's reference implementation, is designed specifically for microcontrollers on various target platforms. Its popularity is evident, with around 2000 different forks on GitHub, each with useful modifications for various development and experimental boards [17]. In practice, using MicroPython is straightforward: its firmware is loaded into the microcontroller's flash memory through standard programming software (often known as a "flash-tool") for the specific type of microcontroller and typically communicates with a computer terminal via a serial interface emulator [18]. A simplified comparison between the architecture of classic development platforms and those based on MicroPython is illustrated in Figure 2 [18].

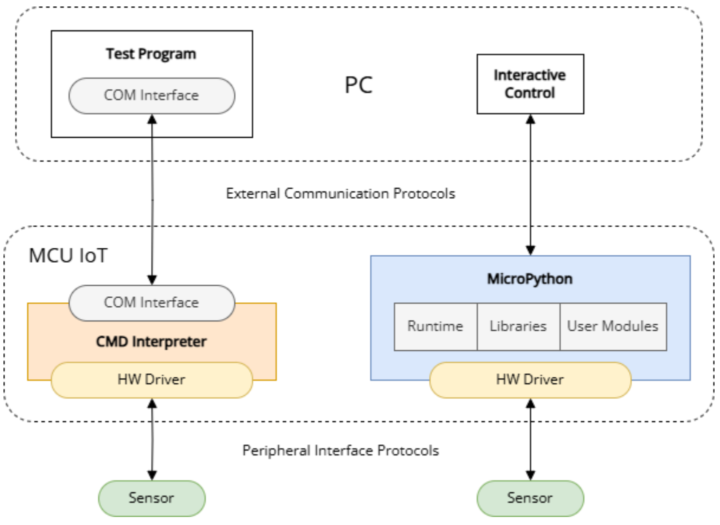


Figure 2. Comparison of standard development platform architecture to one based on MicroPython (adapted from [18]).

MicroPython applications can be categorized into several key areas:

- Teaching and Education: MicroPython enables interactive work with microcontrollers in an REPL (Read–Eval–Print Loop), providing direct access to microcontroller peripherals through any terminal emulator without requiring extensive code for initialization and basic communication. This feature facilitates teaching the fundamental principles of data collection and processing to students across many supported boards, using a straightforward programming language.

- Peripheral and Sensor Development and Testing: MicroPython offers maintained and tested reference implementations for microcontroller interfaces, removing the need for developers to implement the entire communication and control structure for peripheral devices. Modern integrated peripherals communicating over serial interfaces (I2C, SPI, CAN, etc.) are often controlled by reading and writing values from dozens of different registers, each with unique bit meanings and value interdependencies. With simple and widely adopted language interaction, developers can easily verify peripheral functionality, develop and debug the relevant hardware, and create algorithms for device control and data collection. Thanks to hardware abstraction and implementation versatility, other development platforms can be utilized without requiring in-depth programming knowledge of the specific target platform.
- Monitoring and Configuring Complex Applications: Large applications of powerful microcontrollers and FPGAs often incorporate independent monitoring and system parameter configuration tools. For FPGAs, this is commonly a software implementation of a smaller microcontroller, such as the 8051 or Z80, with dedicated programs that enable system parameter monitoring and configuration. Early attempts to implement MicroPython in FPGAs show promise for further development, while in the realm of operating systems, an experimental port exists for monitoring and configuring Linux kernel parameters. For microcontrollers, the situation is simplified with a direct port of MicroPython as a Zephyr application [18].

MicroPython enables code portability across different microcontrollers using a hardware abstraction layer (HAL), while allowing new code additions as libraries (frozen modules) integrated into the firmware to avoid reloading modified code after a microcontroller reset. The base version of MicroPython occupies approximately 20 KB post-compilation, with platform-specific configuration optimizing memory usage by adding standard libraries and peripheral support. Furthermore, MicroPython can be extended with native modules written in C/C++, facilitating low-level peripheral operations and the use of system libraries through interface generators that connect native modules to Python. For microcontrollers with larger FLASH memory, the remaining space can serve as a file system, enabling directory creation and file manipulation [17].

Applications based on this platform in today's microcontrollers provide efficient support for IoT development, enabling robust standard peripheral operations using proven libraries and empowering developers to focus on their unique hardware design, drivers, data conversion, and processing. This also allows the creation of low-level modules for inclusion in final firmware [17]. Altogether, MicroPython has significant potential for future growth, offering extensive documentation and support across numerous development boards, with continuous evolution and an active community poised to secure its position in development tools in the coming years.

2.3. Lua

This VM-based programming language, known for its simplicity and memory efficiency, is ideally suited for embedded virtual environments, particularly for on-demand code execution. In recent years, two main contributions have been made to its application: its use as a VM for embedded parallel operating systems (EPOSs) or as a kernel-level interpreter in operating systems to reconfigure processes and environment variables. The implementation of the Lua Virtual Machine (LVM) in EPOSs represents a significant advancement in enabling high-level programming in resource-constrained embedded systems.

The primary challenge in this endeavour was the inherent incompatibility between the language requirements and the limited capabilities of the target platform. By selectively removing unnecessary LVM features not applicable to embedded applications, a streamlined version was successfully created, retaining essential functions while minimizing overhead. This adaptation process involved a thorough analysis of LVM requirements against EPOS capabilities. Key functionalities, such as character classification, memory management, and basic string handling, were implemented to ensure that the LVM could operate efficiently within the EPOS environment. Notably, typical features found in standard Lua applications, such as software localization and dynamic module loading, were omitted as they were not relevant for these systems. This not only reduced the LVM size to under 155 KB, but also enhanced the execution speed in EPOSs compared to traditional Linux environments. This optimization is crucial for microcontrollers, where memory and processing power are notably limited [19].

The use of LVM in EPOSs allows the execution of Lua applications, which are known for their simplicity and high-level abstractions. By providing a virtual machine capable of interpreting Lua code, EPOSs enable developers to leverage the benefits of high-level programming, including simpler application development and maintenance. A special "Lua Profile" was created with specific LVM-supported functionalities within the EPOS context, serving as a guide for developers to clarify available features and ensure applications are developed without expecting unsupported capabilities. The advantages of this approach are manifold. First, it enables Lua-based application development, which can significantly reduce development time and complexity. Second, the optimized performance of LVM in EPOSs allows applications to run efficiently, making Lua viable in various embedded scenarios. The performance tests conducted showed that while C++ applications executed faster in Linux due to standard library optimizations, Lua applications performed comparably well in EPOSs, confirming the efficiency of this implementation [20].

Implementing the Lua interpreter at the kernel level of operating systems such as NetBSD, as shown in Figure 3 [21], and Linux marks a notable step towards creating scriptable operating systems that allow users to dynamically extend kernel functionality. This is achieved by embedding Lua as a scripting language and writing scripts that can directly interact with the kernel. The approach involves modifying kernel subsystems to invoke the Lua interpreter, giving scripts control over various kernel operations, such as CPU frequency scaling and network packet filtering. This integration not only enhances operating system flexibility, but also facilitates rapid development and experimentation with new kernel algorithms, allowing developers to quickly test and deploy new functions without extensive recompilation or system shutdowns [21].

The benefits of such an interpreter during system reconfiguration are considerable. By allowing users to define custom rules and mechanisms via scripts, the operating system can adapt to specific requirements and changing workloads. For example, a user-defined script can implement a CPU frequency scaling policy that considers not only CPU load but also temperature, thereby preventing overheating while optimizing performance. Additionally, the ability to script complex operations, such as packet filtering, enables more sophisticated network security measures, allowing users to create rules that respond to real-world conditions, such as blocking traffic from vulnerable services [21]. This dynamic scripting capability ultimately leads to more efficient system management and improved responsiveness to user needs, making the kernel-level interpreter a powerful tool for enhancing operating system performance and adaptability.

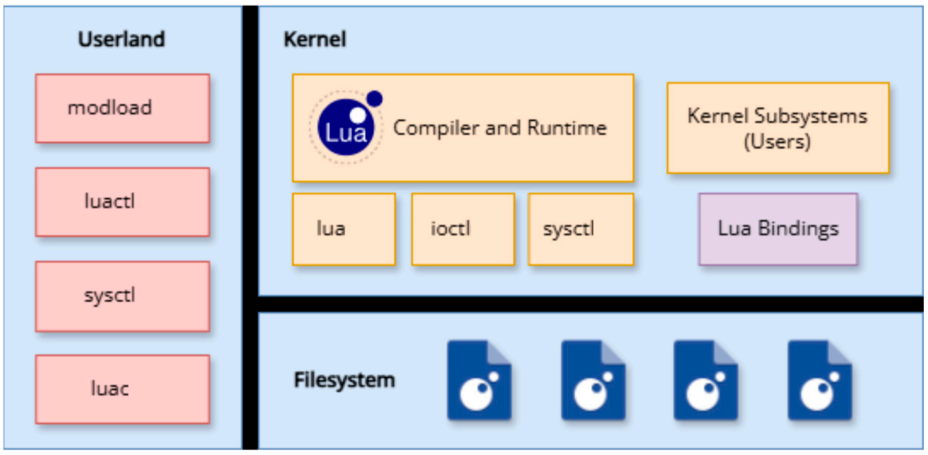


Figure 3. Overview of Lua in the NetBSD kernel (adapted from [21]).

2.4. Examples of Framework Implementations with Dynamic Configurations

The following sections will explore the implementation of the ESP8266 IoT Framework [22,23], a customized framework designed to facilitate the development of IoT applications using the ESP8266 microcontroller. This framework addresses common challenges faced by developers, such as WiFi connection management and HTTP request handling, and provides a user-friendly web interface for configuration. Additionally, the application of a configuration manager will be described in detail, enabling the dynamic management of device settings, along with key components such as web server, WiFi Manager, HTTP request handler, and OTA updates. The Ewings Framework [24], an advanced software package that builds on Arduino ESP8266 libraries, will also be presented, offering additional features and services that simplify IoT application development, especially in the implementation of web services with advanced content filtering and user interface generation.

2.4.1. ESP8266 IoT Framework

The ESP8266 IoT Framework is a custom-built framework designed to facilitate the development of IoT applications using the ESP8266 microcontroller. This framework tackles several common challenges faced by developers, including WiFi connection management, HTTP request handling, and providing a user interface for configuration. It is designed to be unobtrusive and easy to deploy, emphasizing modularity and scalability. The web interface was developed using React, allowing for a responsive and dynamic user experience, while the architecture was envisioned to be flexible enough to adapt to various IoT projects while maintaining a consistent set of functionalities [22].

The framework consists of five primary components structured as in Figure 4:

The WiFi Manager is responsible for connecting the ESP8266 to available WiFi networks. If no known network is detected, it activates a hotspot with a captive portal, enabling users to enter their login and network details. This functionality ensures that the device can connect to the internet without pre-set network names and passwords. After that, the web server component is critical for providing the user interface that allows users to configure settings and monitor device status. It communicates with the ESP8266 via RESTful API, enabling quick and structured interactions between the frontend and the underlying hardware. It also includes a robust HTTP request handler that simplifies the process of creating HTTP and HTTPS requests. This is especially important for IoT applications requiring secure communication with web services. The handler abstracts away the complexities of managing SSL certificates, allowing developers to focus on application logic. The configuration manager is designed to handle user-defined parameters that can be modified while the device is operating. It stores these parameters in EEPROM, ensuring they are retained even when the device is powered off. Parameters are defined in JSON format, simplifying the process of adding or modifying settings. Lastly, OTA updates are a key feature for IoT devices, allowing developers to publish firmware updates without needing physical access to the device. This is particularly useful for devices located in remote or hard-to-reach locations. The framework provides a simple mechanism for users to upload new firmware through the web interface [22].

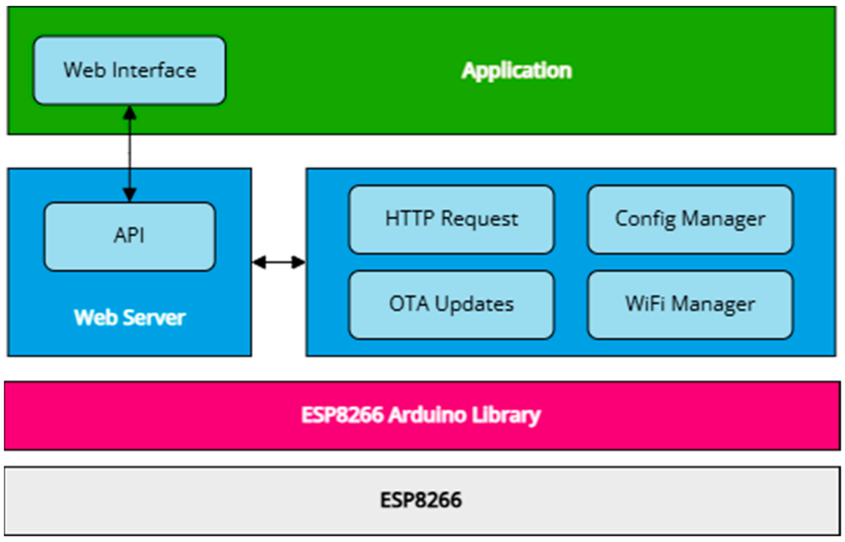
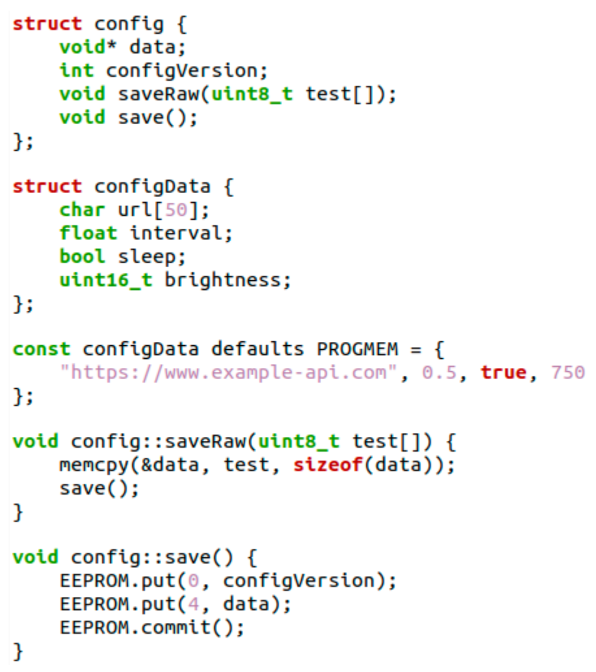


Figure 4. ESP8266 IoT Framework architecture (adapted from [22]).

The configuration manager is a key component of the ESP8266 IoT Framework, designed to facilitate the dynamic management of device settings through an intuitive web interface. This component enables developers to define, modify, and store various parameters governing the behaviour of their IoT applications. Key aspects of its implementation include defining data structures, initializing default values, generating code from JSON into C++, and mechanisms for saving and loading configurations. At its core is a clearly defined data structure, typically represented as a C++ structure. This structure includes parameters such as URLs, data collection intervals, sleep mode triggers, and brightness levels. Default values are set for all parameters, providing a reliable operational foundation. A Python script converts the JSON configuration into C++ code during the compilation process, allowing for straightforward upgrades and modifications while maintaining consistency throughout the application [23].

The persistence of user-defined settings is achieved through functions that save and load configuration data to and from EEPROM (Electrically Erasable Programmable Read-Only Memory), which retains settings even when the device is powered down, an example of which is shown in Figure 5 (implemented as part of this study). When users modify settings through the web interface, these changes are recorded and saved, ensuring that updated configurations are stored and available at the next power-up. The web interface plays a vital role in the configuration manager's functionality, as it is dynamically generated based on the JSON file [23]. This allows users to view and modify configuration parameters in real time. When changes are submitted, the configuration manager processes them anew, enabling seamless interaction that simplifies the process of customizing device settings. This component thus enhances the flexibility and user experience of the ESP8266 IoT Framework, making it easier for developers to create adaptable IoT applications.



```
struct config {
    void* data;
    int configVersion;
    void saveRaw(uint8_t test[]);
    void save();
};

struct configData {
    char url[50];
    float interval;
    bool sleep;
    uint16_t brightness;
};

const configData defaults PROGMEM = {
    "https://www.example-api.com", 0.5, true, 750
};

void config::saveRaw(uint8_t test[]) {
    memcpy(&data, test, sizeof(data));
    save();
}

void config::save() {
    EEPROM.put(0, configVersion);
    EEPROM.put(4, data);
    EEPROM.commit();
}
```

Figure 5. Configuration definition and persistence using EEPROM library functions (authors' contribution).

2.4.2. Ewings Framework

The Ewings ESP8266 Framework is an advanced software package designed to simplify the development of IoT applications using the ESP8266 microcontroller. This framework builds on Arduino ESP8266 libraries, providing a user environment where the full capabilities of the ESP8266, especially its WiFi functionalities and various service integrations, can be fully utilized. At its core, the Ewings Framework leverages the ESP8266EX chip, which features a 32-bit Tensilica L106 processor along with integrated WiFi capabilities. The framework is conceptualized as an operating system-less development tool that provides basic APIs for tasks such as data transmission, TCP/IP functions, and hardware communication. By using existing Arduino libraries, the Ewings Framework aims to make IoT application development more accessible to developers already familiar with the Arduino IDE [24].

Upon initialization, the ESP8266 device creates a local WiFi network, allowing users to connect and configure settings through a web interface. Default access credentials are provided so that users can change configurations such as WiFi settings and access point parameters. This setup process is designed to be simple, ensuring that development teams can quickly deploy their devices. The Ewings Framework offers a variety of services that significantly simplify the development of IoT applications. The HTTP Service extends the Arduino HTTP client, allowing applications to easily make RESTful HTTP requests. The NTP Service provides network time synchronization, ensuring that applications always have accurate time data. The MQTT Service implements the MQTT messaging protocol, facilitating device-to-device communication and sensor monitoring, while the Event Service registers event listeners for specific tasks triggered by events, thereby supporting responsive application behaviour.

The framework also supports OTA firmware updates through its OTA service, offering remote device management and updates without needing physical access to the device. The ESPNOW Service enables mesh networking and broadcasting capabilities, using the ESPNOW feature of ESP8266 for efficient device communication. The WiFi Service provides simplified APIs for dynamic interaction with WiFi devices, including support for internet connections and dynamic subnetting. The PING Service extends basic ping functionality, allowing developers to check active internet connections, and the GPIO Service interacts with GPIO pins for sensor reading and device control, while the MAIL Service uses SMTP to send emails directly from the device, requiring SMTP configuration details. Additionally, the GPIO Alerts Service provides a mechanism for sending alerts based on specific GPIO conditions. All available services, libraries, and utilities are presented in Figure 6 [24].

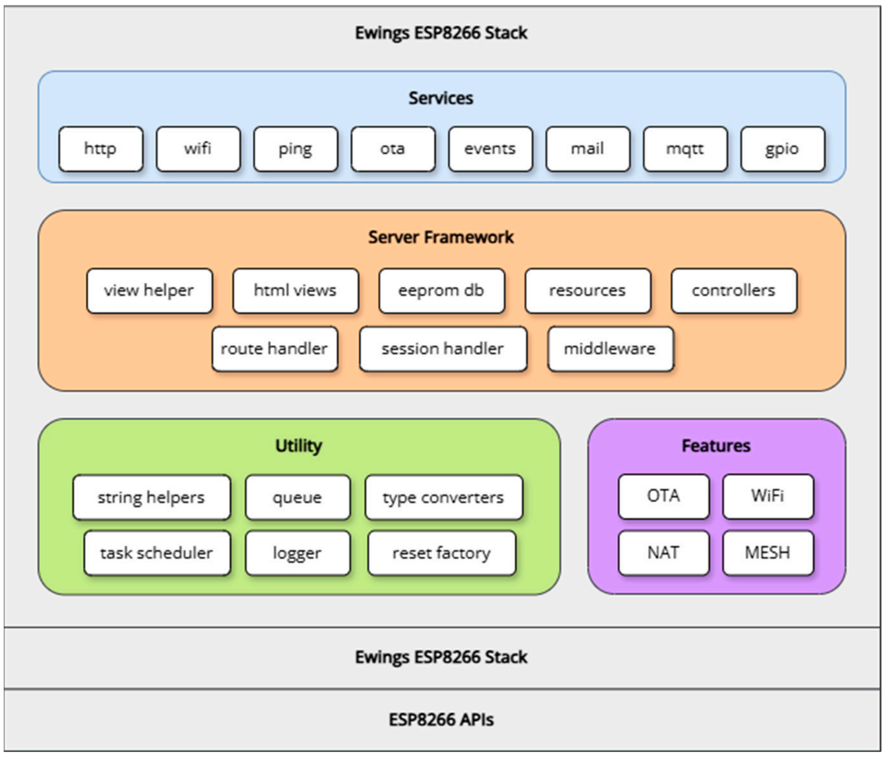


Figure 6. Ewings ESP Framework structure (adapted from [24]).

The Ewings Framework includes an embedded local HTTP server that operates in both station and access point modes. This server facilitates user interactions through a web interface for configuration and device monitoring. Key components of the server include controllers that manage client requests and handle user input, middleware providing request filtering capabilities, and a session manager for user sessions that can expire after a predefined duration. The routing module manages the definition of web routes and authenticates requests, while the EEPROM database uses flash memory to persistently store configuration data. Views and View Helpers assist in dynamically generating HTML content for the web, enhancing user experience [24].

In addition to its core services, the Ewings Framework includes several built-in features that enhance its overall functionality:

- NAT (Network Address Translation): This feature enables IP address mapping, extending the range of networks with active internet connections;
- Mesh Networking: Mesh networking capabilities facilitate the creation of mesh networks using the ESPNOW feature, promoting efficient inter-device communication;
- Background Operation Tools: A collection of tools supporting background operations such as task scheduling, logging, and data-type conversion;
- Device IoT (currently in beta): This feature manages data transmission to an MQTT server, with configurations managed via the local server [24].

The Ewing's Framework can also be suitable for edge computing, especially when considering its compatibility with lightweight, flexible, and distributed applications that need to run close to data sources.

3. Materials and Methods

In this section, the authors describe the chosen development tools and platforms as well as language concepts in detail to develop a proposal for the framework architecture for the dynamic reconfiguration of IoT-enabled embedded systems. The performance of the proposed architecture is tested in Section 4.

3.1. Development Tools and Platforms

The simplified design of the developed framework is shown in Figure 7 below. Using the integrated development environment (IDE), users send configuration data or source code through one of the implemented methods. Primarily, the HTTP protocol and web server are used, which are pre-implemented on ESP32 and ESP8266 boards due to their intuitiveness and ease of application on both ends. Binary instructions are processed on the microcontroller for code integrity verification, restarting the virtual machine instance, and loading new data. Execution is handled within the main loop, which retrieves and processes instructions with each cycle while simultaneously managing potential modification requests.

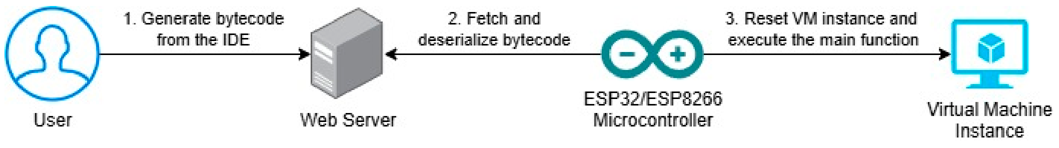

1. Generate bytecode from the IDE
2. Fetch and deserialize bytecode
3. Reset VM instance and execute the main function
User
Web Server
ESP32/ESP8266 Microcontroller
Virtual Machine Instance

Figure 7. Simplified framework workflow overview with pre-installed web server on microcontroller boards (authors' contribution).

The IDE's user interface is implemented using the JavaFX platform in combination with Unirest for handling HTTP requests and the Jackson library for data serialization in JSON format. JavaFX provides a powerful and flexible GUI environment for building modern, responsive IDEs with rich visual components, as well as easy modification and expansion through the MVC approach and design tools like Gluon Scene Builder. Unirest's lightweight HTTP client facilitates the integration of the IDE with external systems and APIs, enabling dynamic reconfiguration, while Jackson's efficient JSON processing allows the IDE to manage complex data structures and configurations that need to be serialized before being sent to the target microcontroller. This combination enables the rapid development of a high-quality IDE that meets the specified design, development, and implementation requirements proposed for this framework [25].

Using available components for data entry and click detection, users also have access to a module for defining the input values for the simulator. In this case, a simple interface for a joystick with directional and action buttons is implemented, which updates the system's state in real time using data binding mechanisms, making them accessible for application in the program code. During simulation runtime, input mapping in the interface to other keyboard buttons is possible for convenience. After the program code is tested in the simulated environment, the modified compiler, implemented in C, can directly execute the code. This is particularly useful for low-level testing without the additional abstraction that the JVM provides, or to create a new bytecode file. Users can then browse available clients, select one, and send that bytecode to the microcontroller for temporary single execution or permanent storage on the file system.

When a code execution request is made, the active virtual machine instance is reset and loaded with the new program code. Two operation modes are available:

- Immediate mode—Upon receiving a request, the new configuration is loaded, and the virtual machine remains active until the end of the program code, after which it finalizes execution (garbage collection and closing of open resources);
- Poll mode—All requests are placed in a separate queue and processed sequentially as the instance is released, keeping the virtual machine active even when the program code has finished executing, waiting for new requests.

The default behaviour is poll mode, assuming that external influences or requests may asynchronously alter the system state, while also keeping the web server active after completing the program sent for configuration. The microcontroller project is implemented using a combination of C and C++ programming languages, achieving broad compatibility regardless of library language. Through the native interface, the virtual environment can connect with available libraries for GPIO, communication protocols, peripheral devices, and other needs, and call them within the bytecode [26]. All implemented modules for the virtual machine, web server, network protocols, file system, and serialization are combined into a single project using PlatformIO, an ecosystem for IoT development that includes a project creation system and a library manager [27]. Within this framework, extensions have also been added for the base project, including serial monitor filters, file system configurations, and additional scripts for file retrieval.

3.2. Language Concepts and Extensions

Stella is a general-purpose language designed to be functional, procedural, object-oriented (class- and prototype-based), and modular. Its development was aimed at creating a flexible and scalable programming environment that supports both high- and low-level procedures and allows for interoperability with languages such as C, C++, and Java. Its simple, intuitive syntax makes it accessible to developers of all skill levels, particularly those with less experience. Programs in Stella are presented as "scripts" containing the necessary declarations for classes, functions, and variables. These scripts can function as standalone modules that execute directly on a virtual machine, without the need for an intermediary language. The language supports class instantiation, function execution, and the manipulation of accessible values, as well as the creation of native functions and structures within scripts [7].

Stella's syntax draws on best practices from languages like C, Java, JavaScript, and Lua, allowing the easy use of conventional statements, expression structures, and control structures, with structures closely resembling those in other dynamic programming languages, with the exception that field modifiers are handled dynamically. Syntax support for variadic arguments and lambda expressions also gives users greater control and flexibility. All expressions execute in the global context of a virtual machine that tracks values of stack, global variables, call frames, and memory management through a garbage collection mechanism [7].

Following adjustments to the original implementation, compiler and interpreter modules are clearly segregated: the compiler can exclusively forward the contents of files or larger code segments and produce bytecode in a separate file, while the interpreter can process the resulting bytecode or execute the source code line by line in REPL mode. This approach was chosen to save memory space in the microcontroller when the system is used solely for code execution (i.e., without the compiler), with REPL mode as an optional feature that can be removed during compilation.

This language is dynamically typed, meaning variables can hold both primitive and reference values. These types are represented by actual values or references, with a type marker that allows the tracking of instance state and type. The basic primitive types include the following:

- Nil: represents uninitialized variables;
- Number: for floating-point values;
- Boolean: for true/false values;
- Object: references to objects or key-value pairs;
- Native: references to native functions [7].

In Stella, objects are represented by a 64-bit header composed of four parts: a 48-bit address pointer to the next object reference (used in garbage collection), a single bit indicating whether the object has been marked within the garbage collector, 4 or more bits to designate the object's type, and a set of bits used as flags for field modifiers and value states. To safely pass objects to functions, each object type must begin with a 64-bit header. In this way, any object type can be cast to an "Object" and, since objects contain their type information, they can be cast back to their original type [7]. Reference types include the following:

- Array: a heterogeneous container containing series of values;
- String: a sequence of characters with auxiliary methods;
- Function: user-defined functions and methods;
- Class: class definitions, including constructors and methods;
- Interface: definitions of methods that need implementation in classes that implement the interface [7].

This type hierarchy is also noticeable in the class diagram of the first prototype implemented in the Java programming language shown in Figure 8. Functions, classes, and interfaces can be stored in variables, passed as arguments to methods, or be used as extensions for other classes and interfaces, allowing new forms of object-oriented programming like traits, mixins, and dynamic inheritance without relying on reflection or dynamic evaluation. When one of these types is created, a new object is added to the value stack containing the name, references to compiled function bodies (or native methods), and additional modifiers and metadata used by the virtual machine during bytecode execution. Each object inherits from the class "Generic", which contains overloaded methods such as "toString" (for converting an instance to a string) and "hashCode" (for returning the object's hash value). Built-in classes already have implementations for these methods, meaning users can leverage existing methods to create hash codes for more complex structures. Objects can also be instantiated as literals (a list of key-value pairs within curly braces) and used as associative arrays (maps or dictionaries). Object fields are stored in a special hash map as strings with standard dot notation (field names), but other key types can be added using index notation. This allows objects that are not bound to user-defined classes to provide the necessary hash map mechanism without sacrificing language reliability and readability. Conversely, arrays serve as heterogeneous containers for storing and iterating over multiple numerically indexed elements. Arrays implement the "Iterable" interface, making it easy for users to iterate through their elements [7].

This approach combines class- and prototype-based programming, allowing developers to establish class hierarchies with shared attributes and behaviours. This approach provides flexibility in defining objects, resulting in unique attributes without changing the base structure. The principles of object-oriented programming, such as abstraction, encapsulation, inheritance, and polymorphism, are supported in Stella. Encapsulation is based not only on data hiding, but also on ownership, where each instance can control the visibility of its components. Polymorphism is achieved through method overriding, while operator overloading allows the creation of new methods with modified markers and name mangling, further enriching the language's functionality [7].

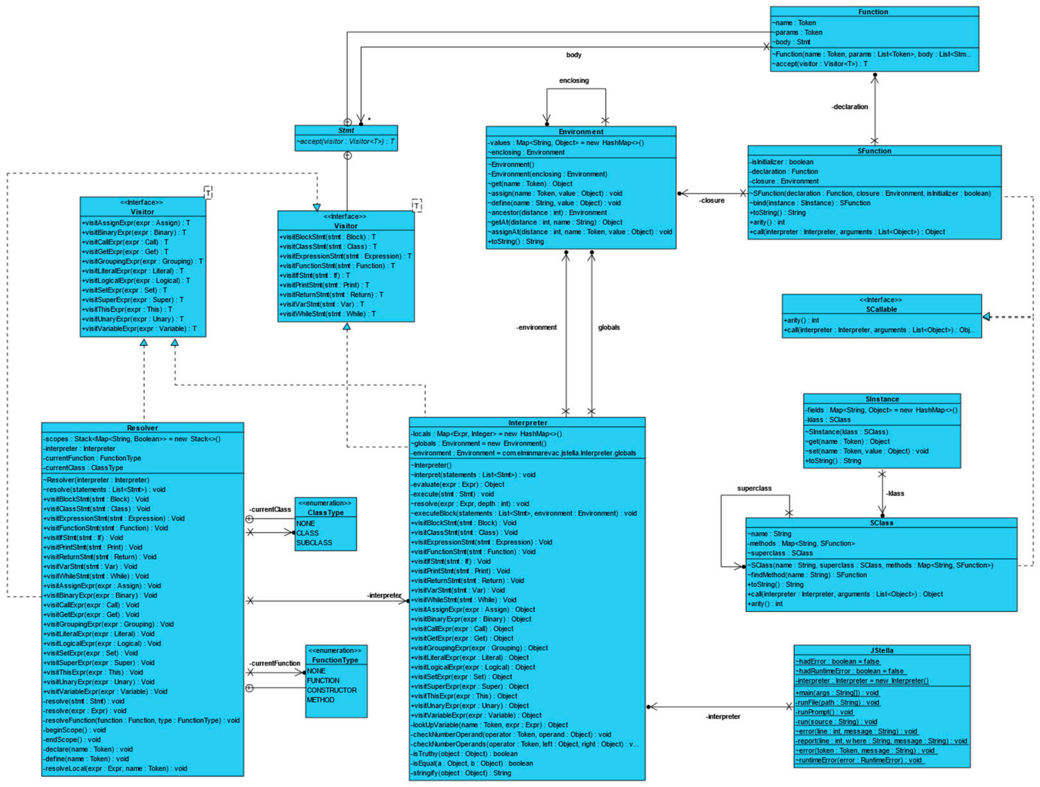


Figure 8. Class diagram of the created prototype (excluding subclasses for expressions and statements) (authors' contribution).

Garbage collection is implemented using the mark-and-sweep algorithm. This algorithm activates after a defined memory threshold (i.e., after allocating a specific number of bytes) and consists of two main steps: marking, which, starting from the root, traverses all accessible objects and marks them as available, and sweeping, which releases all objects unmarked after the marking phase [7].

In the implemented framework for this study, most functionalities are retained, including all control structures, closures, classes, interfaces, static and instance field modifiers, and operator overloading and lambda functions. Support for variadic functions has also been added, as well as strict classes that do not allow the addition or removal of fields after constructor completion, ensuring the persistence of required fields for the native interface and greater reliability during object integrity validation. Using test-driven development for the baseline virtual machine version and auxiliary directives, most of the listed functionalities are modular and can be removed if necessary to conserve memory and improve performance. For example, when executing unary and binary operator instructions, an additional check is introduced for cases where the left operand is an object, which will lead to a search for the appropriate overloaded operator method and invocation with the right operand. If this method invocation mechanism is unnecessary for projects implemented on a microcontroller, development teams can compile a version without it, thereby reducing the number of instructions during operator processing and search time within the method map by eliminating the possibility of signatures for overloading [6].

3.2.1. Available Functionalities in the Standard Library

Predefined language elements allow programmers to access specific data structure elements, manipulate data, and implement design patterns in line with the language specification. Integrated functions are usually designed for tasks that cannot be performed directly from Stella code or the runtime, such as retrieving additional instance-related data, type checking and conversion, and dynamic code evaluation. Libraries created in other languages, such as C or Java, can be easily linked and executed using this API [7]. A standard library for data structures requiring platform-independent implementation, such as maps and lists, is also included as a predefined element, enhancing portability. Additional libraries have been implemented for the following:

- File system access and management, with operations such as directory browsing and file opening, independent of the file system used (LittleFS or SPIFFS);
- Data persistence and configuration in permanent memory, which can be used for storing values within the session or to the file system;
- Network access management, currently supporting connection to WiFi networks;
- Active web server management, defining routes, retrieving parts of the accessed URL or request, and returning responses;
- Mathematical constants and functions;
- Pseudo-random values (via the Mersenne Twister);
- Functions for serial port output and GPIO pin access, such as setting their operating modes, sending and receiving recorded values, and pulse receiving, which can be extended with libraries for targeted components and accessed via the mentioned linking layer.

3.2.2. Compatibility Limitations

In porting the original version of the virtual machine and its libraries, it was essential to account for all limitations, available modules, and the behaviour of memory management functions to ensure an efficient environment that mirrors the functionality of the desktop version. However, due to differences in architecture, platform-dependent library behaviours and calls, as well as resource availability, certain features initially designed for optimization and performance on desktop platforms were either constrained or entirely removed. Furthermore, the handling of requests for loading and processing bytecode required careful consideration, as differences in available data types could potentially lead to corrupted or unexpected states. Below, several constraints are outlined that, while not directly affecting the interpreter's function in the microcontroller, ensure stable platform operation regardless of memory and system requirements.

Initially, Stella's base implementation includes a custom memory management system that categorizes memory fragments into predefined block sizes, reusing freed blocks by storing them in a temporary array for rapid future allocations. This reduces fragmentation and improves performance compared to standard "malloc" and "free" calls. Each allocation in this model includes a memory fragment structure with metadata, allowing efficient management and the quick allocation of available block counts on demand, although this increases overhead since it does not immediately release freed memory but marks it for reuse within the program. This system is primarily designed for caching temporary local variables, such as strings and numbers, with a predefined one-megabyte size [7]. However, given the significantly reduced memory available on microcontrollers, this system was completely replaced with standard system calls and an auxiliary map for caching temporary values. TCP and UDP socket libraries were replaced with integrated network connection libraries, and the threading and signalling mechanism libraries were removed entirely.

The serialization and transmission of bytecode with numeric values, as well as the deep copying of reference types, present additional challenges. Some platforms, such as Arduino Uno, use the "double" label interchangeably with "float" for single-precision floating-point numbers. This can lead to scenarios where values cannot be correctly loaded on certain platforms, and can also result in misaligned bytecodes during deserialization, which may cause unstable behaviour or a corrupted virtual machine state before execution.

To address this, byte sizes were added for storing bytecode values, enabling validation and type determination based on the platform, while compatibility was further enhanced by using 32-bit values for decimal numbers within loading requests. This allows for conversion to 64-bit storage if the virtual machine detects that the format is supported.

Finally, to make memory use more efficient, the garbage collector can be further adapted to the selected development board. The initial collection threshold was reduced from 1 kilobyte to 256 bytes, while the growth factor for subsequent collections remains at 2, meaning the threshold doubles after each collection. An additional parameter, the threshold reduction factor, was added to lower the memory threshold for the next collection in cases where the occupancy criterion is satisfied prior to the next collection. For example, if at any point the virtual machine registers memory occupancy three times below the next collection limit, it can automatically lower the threshold by a predetermined factor. The application of these values depends on the use case and frequency of memory allocation, but they offer development teams greater flexibility in managing collection conditions. For instance, in real-time systems where it is necessary to balance memory usage with time, including collection latency, lower growth and threshold reduction factors will trigger more frequent collections with minimal delay due to fewer allocations. Conversely, in applications with larger and more intensive processing demands, such as batch processing and servers, less frequent collection with a higher threshold can improve throughput by reducing interruptions. Although this approach comes at the cost of slightly higher memory consumption, once load decreases, memory can be freed in a single run, preparing the system for further servicing [5].

3.2.3. Bytecode Serialization

In modern software systems, serialization is a crucial process for preserving program states and data structures, enabling their persistence and transfer between various environments [28]. The initial compiler implementation solely supported the generation of bytecode that could be directly loaded into the virtual machine. To facilitate the transfer of code to a remote network device, a specialized binary format was created for loading onto microcontrollers and for saving values and functions within the file system. This format, with the extension STC (STella Compiled file), encompasses serialization and deserialization processes for any value in the virtual machine, supporting both primitive and reference types through built-in mechanisms for deep copying and the dynamic instantiation of objects and arrays.

Serialization Format

The general structure of the serialization format identifies each value with a type identifier, specifying the data type being serialized, such as numbers, Boolean values, or objects. Following this identifier is the value content, serialized according to its type. For example, primitive types like numbers are serialized as four-byte float values, while Boolean values are serialized as single bytes. Special types like nil, empty, and undefined are represented solely by their type identifier, as no additional data are associated with them. Strings are handled with particular attention: the length of each string is recorded as a two-byte integer, followed by the character sequence of the specified length. Each string is null-terminated to ensure it can be accurately reconstructed during deserialization.

The serialization of instruction sets is more complex and includes several components. Each instruction set includes a line count, serialized as a short integer, followed by the lines themselves, represented by pairs of two-byte values indicating the start and end of each in the instruction array. Additionally, the instruction set includes an instruction count, serialized as a two-byte integer, with the instructions themselves serialized as a byte array.

A key part of instruction set serialization is the value set, which begins with a count of values, followed by the serialized values themselves. This structure allows the easy loading of instruction sets with the corresponding constants within the function, using matching indexes in the instructions and value set.

Objects are serialized with an object type identifier indicating the type, such as functions, lists, or object instances. Each object type has its own serialization requirements. For example, function objects include additional fields such as argument count and enclosed values, serialized as bytes, with their names serialized as strings. Lists have a count of elements and serialized values, while objects contain field counts and pairs of serialized keys and values. Custom serialization methods can be defined for other objects, such as the "Mersenne Twister", which consists of a seed value serialized as an eight-byte integer.

Error Handling

Error handling during deserialization is implemented to ensure robustness. If the end of the content is reached unexpectedly, an error message is displayed, followed by a reset function call to address the error. This mechanism helps maintain the integrity of the deserialization process, preventing incomplete or corrupted data from being reconstructed and loaded onto the virtual machine. Deserialization functions in the code are designed to reverse the serialization process, reading the binary content back into the program's data structures. Each data type is read according to its serialization format, with corresponding structures created and populated with deserialized values. This detailed and structured approach to serialization and deserialization ensures that programs and data can be accurately and efficiently stored and restored, preserving their state and functionality across different executions and environments.

Serialization Process

Figure 9 illustrates the serialization process of a created program, beginning from the function and sequentially passing through its components, recursively serializing all registered values. After program compilation, all data are placed within the main function, which is passed to the serialization function. The main function alone lacks metadata on argument count, enclosed values, and names, as it is implicitly created by the compiler without arguments or a name. After these four values, the process moves to the instruction set, where the line array and instruction offsets are processed first, followed by the array of instructions. To ensure safe serialization, the array length is specified first, followed by its elements, as some arrays, such as the instruction array, cannot be null-terminated due to 0 being a valid virtual machine instruction. Additionally, the element count can be used to terminate the process early if a corrupted file is read, the end of the file is reached, or an unknown instruction/type enumeration is encountered during deserialization.

In serializing the value set, each element begins with its type and associated value for primitive types, while objects require an additional byte for the object type before proceeding to field serialization. It is noteworthy that the compiler exclusively generates the data types shown in the figure, as complex structures like classes, objects, methods, and lists are always evaluated dynamically using the appropriate instructions, which benefits the overall reduction in serialized result size. For instance, this approach would require a more sophisticated handling of the available constant indexes within values or deep copies at each required place in the object, while in instruction set instantiation, values are replaced by instructions that push constants onto the stack by index, occupying only two bytes.

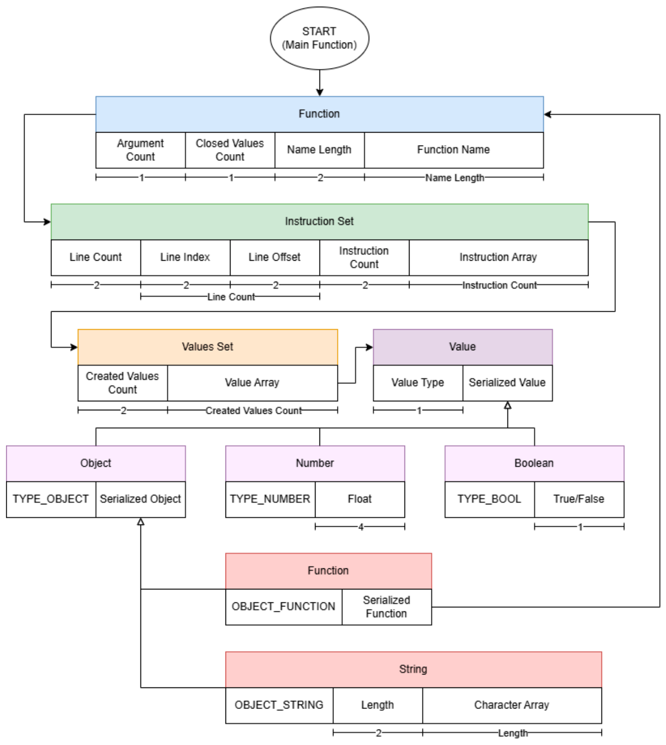


Figure 9. Diagram of the program serialization process (authors' contribution).

3.3. Framework Architecture

In Figure 10, the architecture of the developed framework is illustrated. The client computer, designed to send requests, create new firmware versions, and edit and compile user code, utilizes two main services. The first is an IDE created with JavaFX, consisting of a primary view/scene for reviewing the loaded code/file, a window for viewing the decompiled code, and registered constants before execution. Code compilation is performed using the standard Stella compiler, and within the IDE, messages and bytecode are displayed and loaded via a "pipe" between these two processes. The code can be forwarded to an emulator that includes additional support for modifying the display and defining input components, such as buttons, sliders, and text entry fields. Bytecode transfer is conducted via the HTTP protocol, either sent directly in the request body or using a form with a file field. The user is also offered the option to exchange messages over a UDP port for device discovery on the network, logging, and device status transmission in case of an error or interruption.

The PlatformIO project for the customized virtual machine can be active on any device intended for code transfer to a development board. The most common and reliable method is serial connection via USB, but after the initial firmware transfer, OTA (Over-the-Air) updates are also available, requiring only an IP address, port, and access credentials without any physical connection. The structure and initial contents of the file system are also defined in this project and are sent separately, not being part of the OTA process. The source code for firmware compilation relies on three main artifacts: "ESPSTC" as a library for the virtual machine and native interface, "ArduinoJson" for serializing and deserializing incoming HTTP requests in JSON format, and "platformio.ini", a configuration file for defining the target platform, board, and additional compilation and execution parameters. The main program, implemented as a single-threaded Round Robin architecture with a request queue, consists of two main functions: "setup", which sets up constants, access credentials, initializes I/O and network interfaces, and configures the network server; and "loop", which for each available active module executes the method for request checking and handling. This design not only allows for an even temporal distribution of all services, creating an apparent effect of concurrency, but also enables more efficient request responses without dropping them in cases of resource occupation or the temporary inability to respond due to processing.

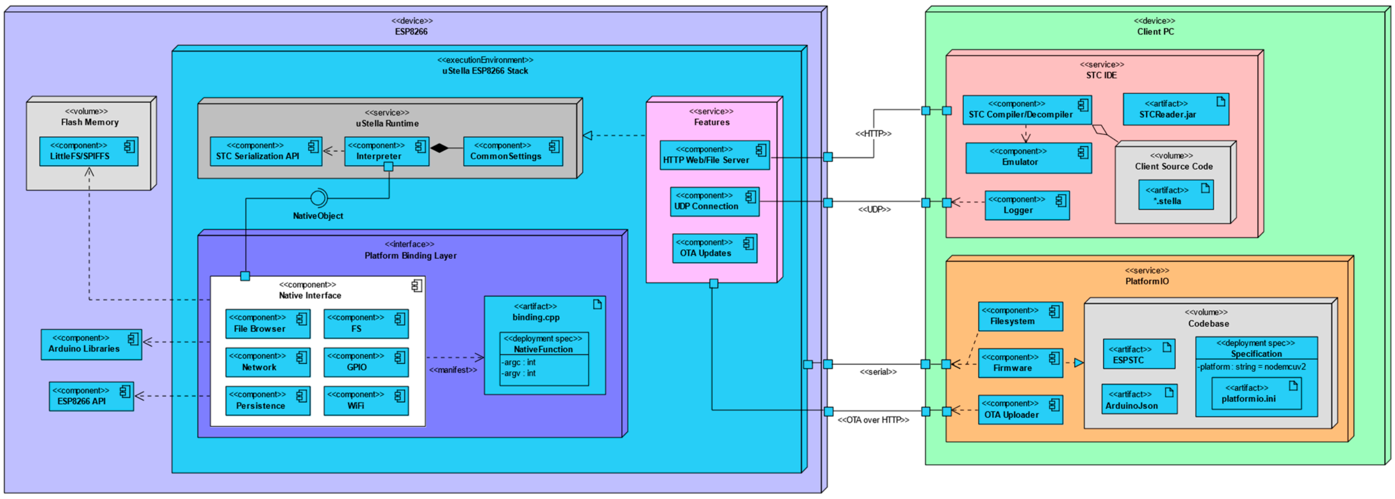


Figure 10. Framework architecture deployment diagram (authors' contribution).

The available system libraries on ESP-based development boards serve as the basis for running identical procedures and memory manipulations as on the computer version of the compiler and interpreter. They also form the foundation for integrating user-accessible services, which, through a platform-independent connection layer to the native interface, manifest as native objects within the instances of the virtual machine. Additional directives are set via configuration headers, while bytecode deserialization is handled by a separate module connected to the interpreter and the data persistence module. Native classes for file access rely on a file system formatted on flash memory using LittleFS or SPIFFS, providing an additional abstraction layer for securely saving files and binary content for serialized values on the stack, and are used in the HTTP server segment intended for file system status inspection [1]. The UDP connection is configurable; by default it is used for locating the development board and sending logs, and it can also be configured to open a WebSocket connection, which would be useful for real-time content updates for web user interfaces [14].

3.3.1. Custom Web Server

The available libraries on ESP platforms allow for the straightforward establishment of wireless network connections, the creation of a web interface for device control and monitoring, the serving of static resources from the file system, and support for other network protocols. However, the core classes have certain limitations, particularly regarding dynamic handler management for HTTP requests and memory management [10]. This has led to the implementation of extensions to support the addition and removal of handlers, direct access to request arguments, and a singleton pattern that ensures the web server is instantiated only once throughout the entire project.

The customized web server retains an identical internal implementation to the base class but inherits and accesses protected fields used for additional functionalities and interaction with the virtual machine:

- The dynamic removal of an existing request handler involves removing the targeted method from the linked list of all registered routes. By default, this method does not automatically deallocate the handler to allow for subsequent addition or on-demand

availability in the firmware, although in user code, it does deallocate memory to conserve resources.
- Access to the server, its methods, and instances is provided through the native “Server” class, which has method signatures identical to those in the “ESP8266WebServer” class. The server is a static resource and is only available as a singleton in native code.
- In active mode, which is preferred for long-term active web servers, incoming requests are queued to be processed after the main user code completes. In immediate mode, on the other hand, requests are ignored, and the virtual machine halts when it reaches the end of the script, requiring users to manually define a waiting state (e.g., using an infinite while loop).
- The base implementation for request parameters exclusively retrieves and checks parameters by index with strict range validation, which will halt the program and reset the microcontroller if unsatisfied. Instead, by accessing the protected attribute for the parameter list, it is possible to implement a method that returns the same result without halting operation, which is then exposed through the native interface.

3.3.2. Route Management Methods

The methods “Server.addRoute” and “Server.removeRoute” are used to add and remove routes via a connection layer to the C++ library for the web server. In the native function, data transfer of the handler occurs through a “RequestHandlerObject” instance with a reference to “RequestHandler” for the web server and a value for the function that will be called in user code. After calling “addRoute” the callback function’s value is transferred to the handler via closure, and upon receiving a request, the procedure adds the callback to the stack, invokes it, and processes its instructions. Instructions are processed until the call stack is emptied, i.e., until the last method with a specified return value exits. The method takes three arguments: the route name, the HTTP method index by the corresponding enumeration, and the callback. If the method is not defined, the default value “HTTP_ANY” is used, meaning the route is not restricted by a specific method that must match in the request.

The returned “RequestHandlerObject” can also be used In user code to invoke the handler or dynamically remove it through the “removeRoute” method, which also deallocates memory for the passed instance, rendering it unusable after the call. If it is necessary to re-register the same callback, creating a new instance with a separate global or lambda function is recommended to avoid code repetition. Removal with a deallocated instance is safe and has no effect.

A greater number of integrated functions for managing parts of HTTP requests and responses are also present in the Server class implementation. When retrieving values, the default return type is a string for compatibility and simplicity. The methods are modelled after standard functions from the ESP8266WebServer class. For example, see the following:

- “header(headerName)” returns the value of a specified HTTP header field (for example, “Server.header(‘Content-Type’)” will return “application/json” if JSON content is sent in the request body).
- “pathArg(i)”, based on the URL parameter index, returns its value. Parameters can be defined in routes using curly braces (for instance, if the route “/items/{}/{}” is defined and accessed via the URL “/items/create/3”, the call “Server.pathArg(0)” will return “create”, while the argument value for “1” will return “3”);
- “inputField(fieldName)” returns the value associated with a form field with the given name or a query string part. If the specified field is not present in either location, the function returns an empty string.

- "send(status = 200, contentType, content)" defines the HTTP response body with the status code, content type, and content itself. For readability, several derived variants of this method were created:
- "plain()" with type "text/plain";
- "json()" with type "application/json";
- "html()" with type "text/html".

3.3.3. Native JSON Format Support

This firmware also includes the implementation of native support for JSON serialization and deserialization. Since the project already relies on the ArduinoJson library as the foundation for JSON handling, a robust and efficient method for parsing and generating JSON data with limited hardware resources is readily available. Deserialization is performed in the connection layer via the C++ code, where key functions are defined for conversion between JSON structures and internal data types. The "fromJson" function takes a string as a value from the virtual machine, deserializes it using the ArduinoJson function "deserializeJson", and then returns a primitive value or recursively structures objects and lists, depending on the parsed type. The "JsonVariant" superclass of all available types in an ArduinoJson document will result in the following:

- Create a number if its type is "double";
- Intern and create an object for a string if the type is "String";
- Create a Boolean value if the type is "bool";
- Create a list that will be populated with values if it is a "JsonArray" instance, where each "JsonVariant" within it is passed again to the "convertToValue" function;
- Create an object with key-value pairs defined from "JsonPair" instances in the "JsonObject" structure, with keys defined as strings and values passed again to the "convertToValue" function.

Integration with the project's virtual machine is achieved through the native function "fromJsonNative", which checks arguments, calls "fromJson", and places the result on the virtual machine stack. If an error occurs during serialization, the function returns null. Conversely, by setting the "PRETTY_PRINT" directive, the direct serialization of any value into JSON format is supported, implemented to enable frequent use for web servers, where a more frequent use of "JsonDocument" instances could lead to excessive dynamic memory usage. Since the virtual machine already has a mechanism for recursive value printing in nested types, it was converted into a function that writes these data into an auxiliary buffer. The buffer's size is estimated using a special function that sums the print length of values, including additional characters and indentations for a more readable format. The process is identical to the deserialization function, except that the structure is traversed via values from the virtual machine.

3.3.4. File System and Server

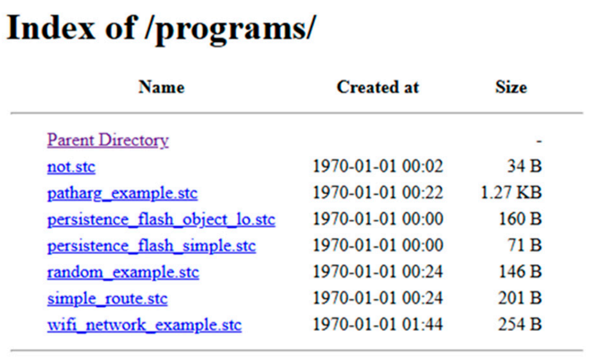
Some system functionalities, such as data persistence, storing static resources, and scripts, require a specific type of persistent memory for storage. The most common method of achieving this is by using formatted flash memory (via libraries such as LittleFS and SPIFFS) or external media (SD cards through the SD library) [1]. Two main routes are registered on the web server for uploading and executing user code from the file system. The first route, "/stc/upload", handles POST requests for file transfers, delegating the actual transfer process to the "handleFileUpload" function, and responds with a simple HTTP status 200 upon successful completion. This function interacts with the HTTP server to manage various stages of the file transfer: initiation, writing, and completion. When the transfer begins (status "UPLOAD_FILE_START"), the file name is checked to ensure it conforms to a specific length constraint (defined by the constant LFS_NAME_MAX from LittleFS, which is 32 characters). If the file name exceeds this limit, it is truncated to fit while retaining the ".stc" extension. The file is then created and stored in the "/programs" directory on the flash memory. As data begin to be written (status "UPLOAD_FILE_WRITE"), the system writes the incoming data blocks to the newly created file. Finally, when the transfer ends (status "UPLOAD_FILE_END"), the file is closed, and the client is redirected to a URL that allows the resource location to be reviewed, signalling a successful upload. If an error occurs while creating the file, the server sends an HTTP 500 response with an error message to the client.

The second route is dedicated to loading a file into the virtual machine and takes the file name as part of the URL (denoted by curly braces {} in the route). It opens the corresponding file from the LittleFS file system. If found, its content is loaded into a buffer and subsequently passed to the "loadFunctionObject" function, which performs deserialization of the bytecode into a FunctionObject. If the virtual machine is already running, it is reset to allow loading a new program. The "initVM" function initializes a new instance, and the function derived from the file is called and interpreted within the main loop. After loading, the server responds with a success message, closes the file, and deallocates the buffer.

The custom "FileAccessHandler" class inherits the "RequestHandler" class and is responsible for handling HTTP requests directed to routes prefixed with "/files". This requires overriding the base implementation of the "canHandle" method, which exclusively checks for the equality of the URL and the defined route. In this case, it verifies if the requested URL starts with "/files", matches it exactly, or ends with a "/", implying that the request pertains to a file or directory within the "/files" prefix. The main component of the class is the "handle" method, which is triggered upon receiving a valid request. This method processes requests for files and directories, distinguishing between file delivery for download and listing directory contents in HTML format. If the request refers to a directory, the method opens the directory using the "lfsOpenDir" function and retrieves its contents with "lfsLs", which returns a linked list of "DirEntry" objects. Both functions are part of the binding layer, making them accessible for user code.

If the request is for a file, the system checks for its existence by calling "LittleFS.open". If the file is found and is not a directory, its content is read into a dynamically allocated buffer and then served to the client as "application/octet-stream" content, suitable for downloads. After delivery, the allocated buffer memory is freed, ensuring efficient memory management in a resource-constrained environment. On the other hand, if the request pertains to a directory, the method dynamically generates an HTML page displaying the directory's contents. The HTML includes a table with the name of each entry, the date of creation, and the size (formatted using the "formatSize" function for files). The list is constructed using multiple calls to the "snprintf" function, creating the HTML structure and inserting directory entries into the table. For directories, the size column shows a dash, while for files, the actual size is displayed, and a link to the parent directory is included at the top of the list [1]. An example of the file browser layout is illustrated in Figure 11 below.

Each file system is preformatted with three directories: "persist" for permanently saving values from the virtual machine, "programs" for programs transferred to the microcontroller, and "static" for static resources. To preconfigure the system, one can generate an image from the project's "data" folder using the command "pio run -t buildfs -e nodemcuv2" and upload it to the microcontroller with the command "pio run -t uploadfs -e nodemcuv2". Registering the handler for the file browser is also straightforward; it needs to be added to the list of available handlers on the server through an instance of "fileAccessHandler" with the "_addNewRequestHandler" method.



Index of /programs/

Name	Created at	Size
Parent Directory		-
not.stc	1970-01-01 00:02	34 B
patharg_example.stc	1970-01-01 00:22	1.27 KB
persistence_flash_object_lo.stc	1970-01-01 00:00	160 B
persistence_flash_simple.stc	1970-01-01 00:00	71 B
random_example.stc	1970-01-01 00:24	146 B
simple_route.stc	1970-01-01 00:24	201 B
wifi_network_example.stc	1970-01-01 01:44	254 B

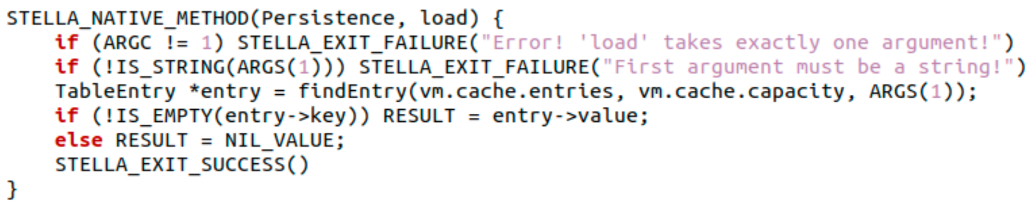
Figure 11. Default file browser layout with placeholder dates used for illustrative purposes.

3.4. Data Persistence

In the context of embedded systems, data persistence refers to the ability to store and utilize data beyond the lifespan of a single program or microcontroller operation. This concept is crucial for state preservation, as it enables three main functionalities: transferring the state between successive instances of a virtual machine, the permanent storage of configurations or globally accessible values, and state recovery in the event of a failure or microcontroller interruption. The implementation framework includes a class called "Persistence" which is used for this purpose and provides a simple programming interface for accessing data at the session level or across microcontroller power cycles. The following sections will elaborate on these two concepts and how they are implemented within the system [28].

3.4.1. Session-Level Data Persistence

The term "session" refers to the finite period of active microcontroller operation that may involve the creation of virtual machine instances [29]. In its standard implementation, the use of a single hash map for global variables is envisioned, which is beneficial for centralized access to not only these variables, but also the available native classes and functions. However, the values within this data structure are erased when the VM's execution ends, as the garbage collector returns the execution environment to its initial state. To address this, an additional hash map named "cache" is added to the global instance of the VM, mapping the names and values of persisted records. Access to this map is facilitated through the methods "Persistence.load(name)" and "Persistence.save(name, value)", as defined with code snippets in Figures 12 and 13. If a record does not exist when loaded, the method returns null, whereas saving an existing key will replace the old value with the new one.



```
STELLA_NATIVE_METHOD(Persistence, load) {
    if (ARGC != 1) STELLA_EXIT_FAILURE("Error! 'load' takes exactly one argument!")
    if (!IS_STRING(ARGS(1))) STELLA_EXIT_FAILURE("First argument must be a string!")
    TableEntry *entry = findEntry(vm.cache.entries, vm.cache.capacity, ARGS(1));
    if (!IS_EMPTY(entry->key)) RESULT = entry->value;
    else RESULT = NIL_VALUE;
    STELLA_EXIT_SUCCESS()
}
```

Figure 12. Implementation of the "Persistence.load" method via the native interface.

A potential issue with executing any method from the "Persistence" class is related to saving external references, such as closed values in functions and methods within classes. Since functions are statically compiled, their access to variables is also static and index-based, which may not align in subsequent executions, potentially destabilizing the virtual machine. To resolve this, the function "checkContainsExternalReferences", a code snippet of which is presented in Figure 14, verifies whether a value with external references is being saved, accommodating complex types like lists or instances by enabling a deep inspection of object structures. Technically, it would be feasible to save function values as long as they lack closures or access to global variables, which would require checking the set of instructions for "OP_GLOBAL" bytecodes, a feature that has not been implemented.

```
STELLA_NATIVE_METHOD(Persistence, save) {
    if (ARGC != 2) STELLA_EXIT_FAILURE("Error! 'save' takes exactly two argument!")
    if (!IS_STRING(ARGS(1))) STELLA_EXIT_FAILURE("First argument must be a string!")
    if (!checkContainsExternalReferences(ARGS(2)))
        STELLA_EXIT_FAILURE("Values that contain closures or potential references to external variables cannot be persisted!")
    addEntry(&vm.cache, ARGS(1), ARGS(2));
    STELLA_EXIT_SUCCESS()
}
```

Figure 13. Implementation of the "Persistence.save" method via the native interface.

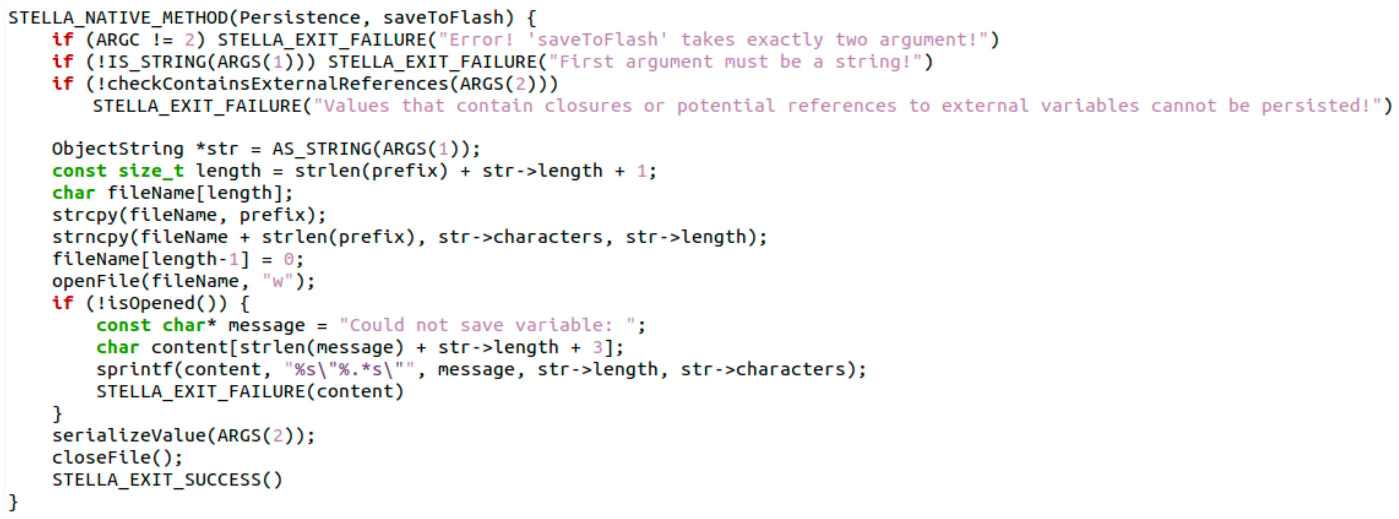
```
static bool checkContainsExternalReferences(Value value) {
    if (!IS_OBJECT(value)) return true;
    Object* obj = AS_OBJECT(value);
    switch (getType(obj)) {
        case OBJECT_METHOD: case OBJECT_CLASS: case OBJECT_CLOSURE:
        case OBJECT_FUNCTION: case OBJECT_REQUESTHANDLER: return false;
        case OBJECT_LIST: {
            ListObject *list = (ListObject*)obj;
            for (uint32_t i = 0; i < list->counter; ++i)
                if (!checkContainsExternalReferences(list->items[i])) return false;
            return true;
        }
        case OBJECT_INSTANCE: {
            InstanceObject *instance = (InstanceObject*)obj;
            Table* table = &instance->fields;
            for (uint32_t i = 0; i < instance->fields.capacity; ++i) {
                if (!checkContainsExternalReferences(table->entries[i].key)
                || !checkContainsExternalReferences(table->entries[i].value))
                    return false;
            }
            return true;
        }
        case OBJECT_CLUSTERVALUE: {
            ClusterValueObject *clusterValue = (ClusterValueObject*)obj;
            while (clusterValue != NULL) {
                if (!checkContainsExternalReferences(OBJECT_VALUE(clusterValue))) return false;
                clusterValue = clusterValue->next;
            }
            return true;
        }
        case OBJECT_STRING: case OBJECT_MERSENNETWISTER: case OBJECT_NATIVE: return true;
    }
    return true;
}
```

Figure 14. Implementation of the "checkContainsExternalReferences" function.

Ordinarily, invoking the garbage collector would delete all values from the "cache" map if they are not referenced in the code. To prevent this, all saved values in the map are marked before the garbage collection cycle, ensuring they are excluded from the cleaning phase. This approach is also used to reset the virtual machine state, which should retain the map's contents while deallocating all other values created during program execution. The map keys must also be interned and ignored to allow consistent access, as each VM reset clears the string interning map. Given that the equality check for strings relies on this mechanism, failing to intern keys would result in the user being unable to find records, as the key and the string in the code would reference different objects.

3.4.2. Long-Term Storage on Device Flash Memory

For saving files on the file system, 1 MB of flash memory is available, sufficient for smaller text files, static resources, and serializing values from the VM. The methods "Persistence.loadFromFlash (name)" and "Persistence.saveToFlash (name, value)" leverage this storage method for values that need to remain accessible after the microcontroller shuts down, invoking the compiler's serialization and deserialization methods, respectively, using the key as the name of the newly created file. To avoid name collisions between user files and serialized values, all serialized data are stored in a dedicated "persist" folder. Methods from LittleFS are called directly through the interface layer using functions like "openFile", "isOpened", and "closeFile" to handle the file descriptor, and "serializeValue" and "deserializeValue" for reading and writing file content. The functions for serialization and deserialization interfacing with the native layer are identical to those in the compiler, relying on predefined methods and the structure that has already been described. An implementation of the "Persistence.saveToFlash" method is shown in Figure 15, whereas the method for loading the value from flash memory uses the read mode for opening the file and invokes the "deserializeValue" function of the active file descriptor.



```
STELLA_NATIVE_METHOD(Persistence, saveToFlash) {
    if (ARGC != 2) STELLA_EXIT_FAILURE("Error! 'saveToFlash' takes exactly two argument!")
    if (!IS_STRING(ARGS(1))) STELLA_EXIT_FAILURE("First argument must be a string!")
    if (!checkContainsExternalReferences(ARGS(2)))
        STELLA_EXIT_FAILURE("Values that contain closures or potential references to external variables cannot be persisted!")

    ObjectString *str = AS_STRING(ARGS(1));
    const size_t length = strlen(prefix) + str->length + 1;
    char fileName[length];
    strcpy(fileName, prefix);
    strncpy(fileName + strlen(prefix), str->characters, str->length);
    fileName[length-1] = 0;
    openFile(fileName, "w");
    if (!isOpened()) {
        const char* message = "Could not save variable: ";
        char content[strlen(message) + str->length + 3];
        sprintf(content, "%s\"%.*s\"", message, str->length, str->characters);
        STELLA_EXIT_FAILURE(content)
    }
    serializeValue(ARGS(2));
    closeFile();
    STELLA_EXIT_SUCCESS()
}
```

Figure 15. Implementation of the "Persistence.saveToFlash" method via the native interface.

4. Results

This section presents the results of various tests conducted by the authors to evaluate the performance of the proposed framework architecture. The analyses include performance comparisons, garbage collector stress tests, serialization and deserialization evaluations, and software update comparisons, with detailed discussions of the findings. It is important to note that the benchmarks and stress tests were designed specifically for this study to evaluate the performance of the proposed framework under various conditions.

The following section presents the results of the benchmark tests conducted to evaluate system responsiveness and throughput during garbage collection and the time required for updates via OTA requests, serial connections, and firmware implemented with a virtual machine in response to HTTP requests. The development board used for running these update and garbage collection tests was "NodeMCU ESP8266 Amica V2", configured with a total of 4 MB of flash memory, of which 1 megabyte was allocated for the internal file system and approximately 1 megabyte (~1019 KB) for OTA updates. The remaining memory was used for allocations and processing web requests within the web server class, satisfying the requirement that the OTA update space be at least twice the size of the firmware, which is estimated to be around 302 KB. Additionally, the tests for NodeMCU disregarded the time needed to output board status messages or request content over the serial connection due to a fixed transmission speed (baud rate) of 9600 bps [30].

Since each character is represented in a 10-bit format without a parity bit (comprising 1 start bit, 1 stop bit, and 8 bits for the ASCII value) the time required to output one character is in the order of milliseconds, which can significantly increase the request processing time for larger content or when printing additional messages, regardless of the running procedure. The benchmark tests use an identical configuration to those of the Lua 5 programming language, with a setup virtualized to achieve 512 MB of RAM and a single-core processor running at a clock speed of 2.0 GHz on Linux 2.6 [31].

4.1. Performance Comparison

Table 1 presents the results of executing certain test cases from "The Great Win32 Computer Language Shootout" suite, which were adapted from the original code for Visual C++ [32]. Two programming languages are included for comparison. The first is Lua, represented by three versions to illustrate the differences in implementation and optimization levels:

- Lua 4.0 implements a stack-based virtual machine similar to Stella, utilizing only tables for storing objects and arrays;
- Lua 5.0 adopts a register-based approach, potentially leading to a more efficient use of load and save instructions, shorter bytecode, and faster execution due to fewer operations on values compared to pushing and popping them from the stack;
- Lua $5'$ represents a variant of Lua 5.0 without table optimizations for arrays, tail recursion, or dynamic stacks [31].

Table 1. Execution time comparison of Lua, Stella, and MicroPython (measured in milliseconds).

Program	Lua 4.0	Lua $5'$	Lua 5.0	Stella	Python 3.4
sum (2×10^7)	1.237	0.541	0.546	1.622	1.481
fibonacci (30)	0.953	0.684	0.690	0.080	0.139
ack (8)	1.004	0.863	0.884	0.123	0.238
random(1×10^6)	1.045	0.969	0.963	0.121	0.168
sieve(100)	0.937	0.827	0.572	0.195	0.192
heapsort (5×10^4)	1.086	1.050	0.703	0.081	0.109
matrix (50)	0.847	0.828	0.599	0.122	0.097

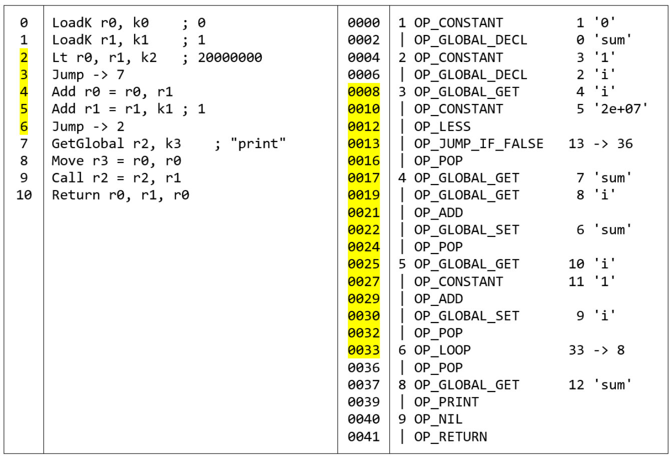
On the other hand, the selected version of MicroPython implements Python 3.4 along with select extensions from more recent language versions while retaining the core features of the CPython interpreter [17]. MicroPython was executed using Unix version port v1.10 in the same configuration as the other languages. The results presented in this table are derived from the average of twenty executions.

The first test, involving summing a large number of values, clearly indicates the advantage of using registers for executing mathematical operations efficiently. The examination of the compiled bytecode shows that Lua 5.0 successfully maintains these values in two separate registers, with sum addition and counter incrementation written as two addition instructions. In contrast, stack-based languages must push both values onto the stack for each operation, execute the operation, pop the value from the stack, store it in a variable, and repeat this process for every computation and loop iteration. Decompilation further highlights this, as Lua 5.0 features a five-instruction loop compared to Stella's sixteen instructions, approximately proportional to the difference in execution time between the two languages, as highlighted in Figure 16.

For recursive functions, such as the Ackermann or Fibonacci series, Stella demonstrates significant acceleration for two main reasons:

- The interpreter uses an array of call frames for caching less deeply nested functions, doubling the frame size for deeper recursion. Conversely, Lua and Python allocate a separate object to track each call frame, deallocating it upon removal. This allocation overhead is evident, particularly in the Ackermann function, which runs almost twice as slowly in Python.

- Each call records the index of the new frame's starting value on the stack, which also serves as the call's return value. Consequently, access to local variables is achieved through this starting index and the instruction is offset without duplicating values, while object marking is determined solely by the stack's top index.



```
0    LoadK r0, k0    ; 0
1    LoadK r1, k1    ; 1
2    Lt r0, r1, k2   ; 20000000
3    Jump -> 7
4    Add r0 = r0, r1
5    Add r1 = r1, k1 ; 1
6    Jump -> 2
7    GetGlobal r2, k3    ; "print"
8    Move r3 = r0, r0
9    Call r2 = r2, r1
10   Return r0, r1, r0
```

```
0000  1 OP_CONSTANT         1 '0'
0002  | OP_GLOBAL_DECL      0 'sum'
0004  2 OP_CONSTANT         3 '1'
0006  | OP_GLOBAL_DECL      2 'i'
0008  3 OP_GLOBAL_GET       4 'i'
0010  | OP_CONSTANT         5 '2e+07'
0012  | OP_LESS
0013  | OP_JUMP_IF_FALSE   13 -> 36
0016  | OP_POP
0017  4 OP_GLOBAL_GET       7 'sum'
0019  | OP_GLOBAL_GET       8 'i'
0021  | OP_ADD
0022  | OP_GLOBAL_SET       6 'sum'
0024  | OP_POP
0025  5 OP_GLOBAL_GET      10 'i'
0027  | OP_CONSTANT        11 '1'
0029  | OP_ADD
0030  | OP_GLOBAL_SET       9 'i'
0032  | OP_POP
0033  6 OP_LOOP            33 -> 8
0036  | OP_POP
0037  8 OP_GLOBAL_GET      12 'sum'
0039  | OP_PRINT
0040  9 OP_NIL
0041  | OP_RETURN
```

Figure 16. Comparison of the decompiled code for Lua 5.0 and Stella with loop instructions highlighted (authors' contribution).

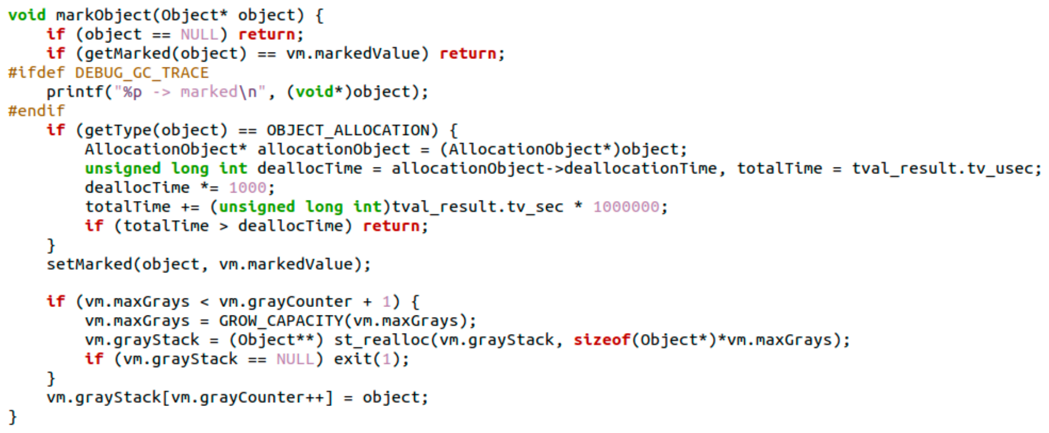
The Linear Congruential Generator (LCG) function employs a comparable number of instructions across languages but benefits from avoiding the constant allocation and deallocation of call frames when fewer calls are made. The use of primitive single-precision floating-point types also prevents the instantiation of new number objects, unlike Python. The final three functions for Eratosthenes' sieve, heapsort, and matrix operations rely on single and multidimensional array behaviour. Their performance is comparable to Python's since the underlying data structure implementations are identical, using variable-length array lists for storing values/objects. However, prior to version 5.0, Lua did not have an equivalent structure. Instead, tables were used, functioning as hash maps with integer keys, requiring hash calculations for each operation. The insertion or removal of elements could also trigger rehashing using a chained scatter table with Brent's variation, all of which could lead to performance drops. From version 5.0 onwards, a hybrid approach with separate arrays for hashed values and list elements yielded noticeable performance improvements [31], albeit not on par with other languages.

The results highlight Stella's efficiency in handling recursive and stack-based operations due to its optimized use of call frames and minimal overhead in local variable access. However, Lua 5.0's register-based approach clearly outperforms older stack-based versions in mathematical operations and loop execution. While Python and MicroPython maintain reasonable performance in array and list-based operations, their object-oriented memory handling introduces delays in recursive scenarios. Stella's overall performance remains comparable to other languages and is particularly suitable for microcontroller programming because of its lightweight stack-based design, which minimizes resource usage and efficiently manages memory—a critical requirement in embedded systems with limited computational power [33].

4.2. Garbage Collector Stress Testing

Efficient memory management is critical in embedded system development, particularly when deploying a virtual machine (VM) on microcontrollers with constrained hardware resources. Microcontrollers like NodeMCU ESP8266 have limited RAM and flash memory, presenting a unique challenge to developers who must optimize performance while ensuring system stability. The effective management of memory resources is essential, as this directly influences system responsiveness, reliability, and the ability to handle concurrent tasks or execute scripts efficiently. When a VM is implemented on a microcontroller, an additional layer of abstraction is introduced, complicating the memory allocation and deallocation processes. Unlike standard computing platforms, where memory resources are relatively abundant, embedded systems require the garbage collector to be highly efficient to prevent performance degradation. The demand for optimized garbage collection is especially pronounced during high-stakes scenarios, such as handling web server requests, performing firmware updates, or managing real-time operations. Any delay in garbage collection can lead to bottlenecks, increased latency, or even system failures if memory runs out during critical tasks [8]. Consequently, garbage collector stress tests are essential for evaluating the system's ability to efficiently manage memory allocation and deallocation under varying loads, simulating realistic conditions such as simultaneous web requests and VM operations.

The performed stress test relies on allocation frequencies, sizes, and object retention times, defined using normal distribution and created for the VM instance as indicators for when a new object should be allocated, with byte number and the timestamp after which the object should be deleted by the GC. The mean value, standard deviation (sigma), and bounds are carefully defined to emulate typical allocation behaviours in an embedded environment. Specifically, the allocation frequency measured in seconds has a mean of 0.08 with a sigma of 0.03, without upper bounds, indicating a moderate allocation rate with some variability. Allocation sizes follow a distribution with a mean of 64 bytes and a sigma of 32 bytes, constrained by a minimum size of 16 bytes to reflect common memory usage patterns. The duration parameter, with a mean of 1 and a sigma of 2, has a minimum value of 0.1, simulating the typical retention time of allocated objects. During the tests, which ran for 20 min, the initial collection threshold was set at 1 KB. Memory allocation entries were imported as an array, where each entry created an instance of the "AllocationEntry" class. These instances were processed in the main loop using the integrated "millis()" function. When the current time matched the allocation time, a new object with the specified size was allocated. The garbage collection mechanism, upon reaching the next collection threshold, performed a marking phase that only marked objects whose retention duration had not yet expired. Consequently, only expired objects were deallocated, optimizing the garbage collection process. The implementation in Figure 17 illustrates an approach more suitable for Linux-based systems using the "timeval" struct from the "sys/time.h" library.



```c
void markObject(Object* object) {
    if (object == NULL) return;
    if (getMarked(object) == vm.markedValue) return;
#ifdef DEBUG_GC_TRACE
    printf("%p -> marked\n", (void*)object);
#endif
    if (getType(object) == OBJECT_ALLOCATION) {
        AllocationObject* allocationObject = (AllocationObject*)object;
        unsigned long int deallocTime = allocationObject->deallocationTime, totalTime = tval_result.tv_usec;
        deallocTime *= 1000;
        totalTime += (unsigned long int)tval_result.tv_sec * 1000000;
        if (totalTime > deallocTime) return;
    }
    setMarked(object, vm.markedValue);

    if (vm.maxGrays < vm.grayCounter + 1) {
        vm.maxGrays = GROW_CAPACITY(vm.maxGrays);
        vm.grayStack = (Object**) st_realloc(vm.grayStack, sizeof(Object*)*vm.maxGrays);
        if (vm.grayStack == NULL) exit(1);
    }
    vm.grayStack[vm.grayCounter++] = object;
}
```

Figure 17. Object marking for time-constrained allocation entries.

The results, as presented in Figures 18 and 19, indicate that the garbage collector handled a total of approximately 957 KB of memory with an allocation rate of around 798 bytes per second and a frequency of 25 objects per second. The modified algorithm demonstrated stable performance and maintained a steady memory footprint, averaging 8.639 KB of used heap memory. The high throughput, calculated using Equation (1), reinforces this observation:

$$T = \frac{TotalTime - GCPauseTime}{TotalTime} \times 100 = \frac{1200\text{s} - 28324\ \mu\text{s}}{1200\text{s}} \times 100 \cong 99.99763\% \quad (1)$$

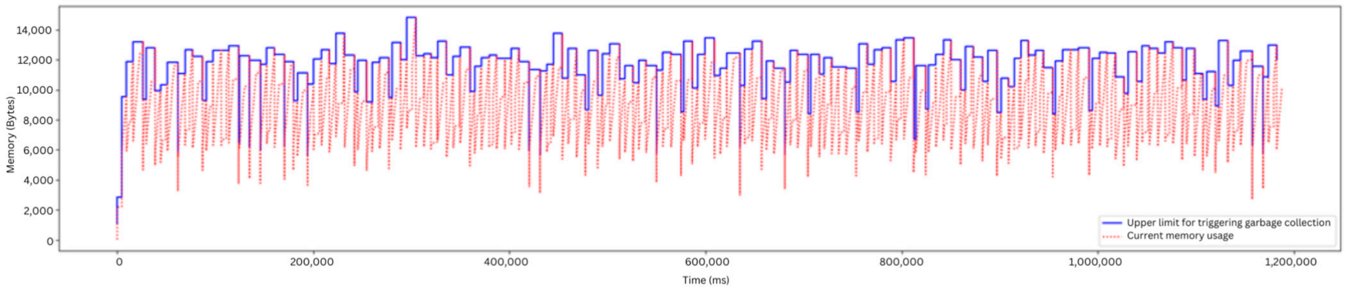


Figure 18. Memory usage with the next collection limit during garbage collector stress testing (authors' contribution).

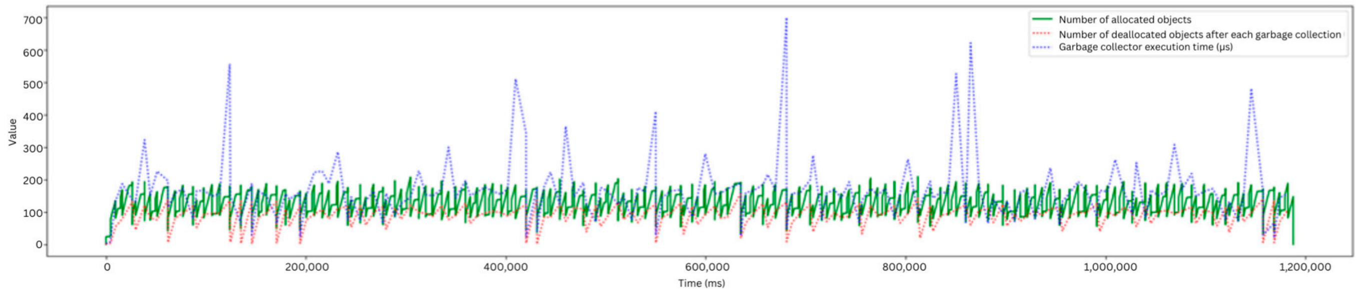


Figure 19. Changes in the number of allocated objects and execution times during garbage collector stress testing (authors' contribution).

This near-perfect throughput suggests that garbage collection had minimal impact on system performance, even under conditions of high object allocation. Nevertheless, several execution time spikes were observed, attributed to either adjustments in the next collection threshold or intense periods of object deallocation. These spikes had a mean execution time of approximately 163 microseconds. Further analysis of the total collection time divided by the number of objects allocated yielded an average of 0.9425 microseconds per object, a performance metric that aligns well with the requirements for real-time applications. However, some of the observed slowdowns during peak periods could pose a bottleneck, necessitating further optimizations to maintain system responsiveness under heavy load [13].

In conclusion, the garbage collection mechanism implemented for the NodeMCU ESP8266 VM proved efficient, with minimal performance degradation during extensive object allocations. The analysis reveals a promising throughput of nearly 100%, suggesting that the system is capable of sustaining high allocation rates without significant latency. Nonetheless, occasional execution spikes highlight areas for potential optimization, especially for applications requiring stringent real-time performance. Future work should focus on refining garbage collection algorithms to further mitigate these bottlenecks, ensuring even greater system stability and responsiveness under demanding conditions.

4.3. Serialization/Deserialization Stress Testing

Serialization and deserialization are crucial processes within the virtual machine, enabling the conversion of function objects to binary data and back, respectively. These operations play an essential role in ensuring data persistence, where serialization allows for values to be saved and retrieved efficiently, and deserialization facilitates the reloading of data into the virtual machine's environment. Particularly in scenarios involving reconfiguration requests, deserialization becomes vital as it interprets new configurations by transforming the binary data into executable functions [28]. This transformation is fundamental for updating the virtual machine's behaviour on the fly, ensuring it adapts seamlessly to new operational requirements.

To evaluate the performance and reliability of these processes, a stress test was conducted on a test suite containing 294 tests of varying complexity, each executed four times. The results, as demonstrated in Figure 20, show that the serialization times are generally consistent, with a median processing time of 160 microseconds. The worst-case scenario reaches 8.357 milliseconds, which remains within an acceptable range for most real-time applications. Even for larger data loads, where tests approach the 800 KB size limit, the observed serialization speed is approximately 10.446 microseconds per kilobyte. This performance should adequately support real-time behaviour updates and system changes, meeting the demand for prompt data preservation under typical conditions.

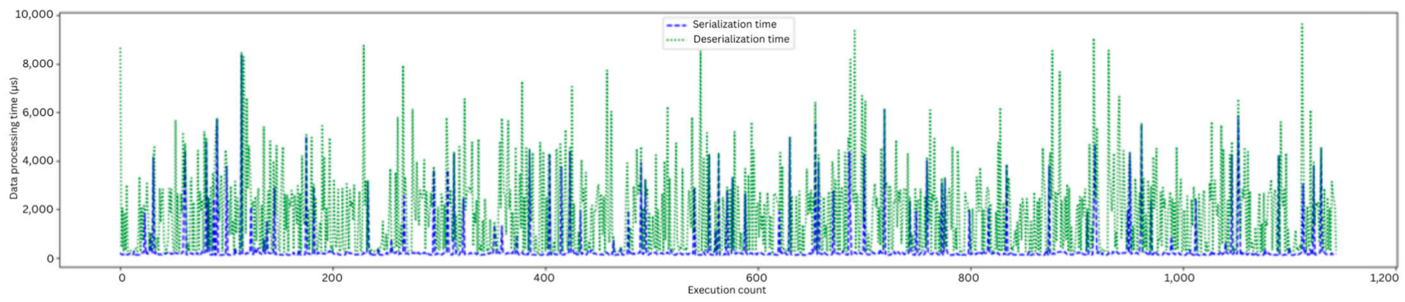
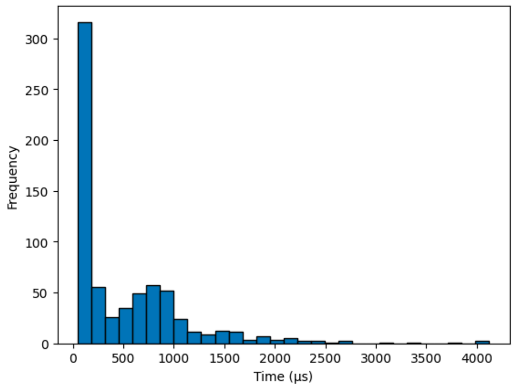
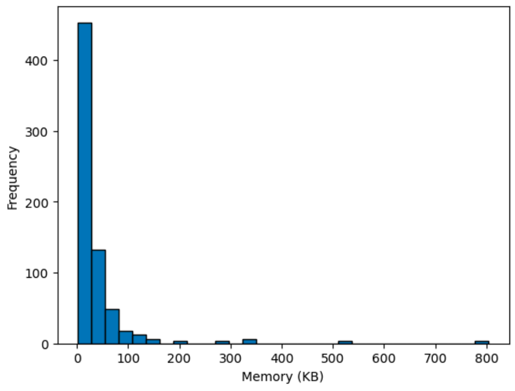


Figure 20. Serialization and deserialization times during stress testing (authors' contribution).

Deserialization, a process with arguably higher significance due to its interaction with new configurations, shows a median completion time of 964 microseconds and a worst-case scenario of 9.396 milliseconds. This elevated time requirement can likely be attributed to the additional tasks involved in deserialization, such as the allocation of new objects for functions and values. Furthermore, deserialization involves potential dynamic array resizing, particularly when initializing large pools for global variables or value maps. Although garbage collection could influence this process by deallocating resources, it is often deactivated during deserialization to prevent the premature removal of newly allocated structures, thus prioritizing stability over memory efficiency during the loading phase. Additionally, the two histograms provided in Figure 21 depict distinct distributions for time (in milliseconds) and memory usage (in kilobytes) during the serialization process. Both distributions are notably right-skewed, with most values concentrated at the lower end of the spectrum. The time distribution shows a pronounced peak between 0 and 500 microseconds, indicating that the majority of serialization tasks are completed rapidly, with only a few outliers extending up to several milliseconds. Similarly, the memory distribution reveals that most serialization tasks consume minimal memory, with the majority falling below the 50 KB threshold, while only a small number of cases exhibit higher memory demands. This skewness in both distributions underscores the efficiency of typical operations in terms of both execution speed and memory consumption, though the presence of long-tail outliers suggests certain high-complexity scenarios that require optimization to minimize resource usage.



(**a**)



(**b**)

Figure 21. (**a**) Histogram for serialization times; (**b**) histogram for serialization request sizes (authors' contribution).

These results are significant as they highlight the virtual machine's operational efficiency under standard conditions, where both serialization and deserialization remain well within real-time processing requirements. The similarity in the distribution shapes for time and memory usage also suggests a potential relationship between these metrics; however, further statistical analysis would be necessary to ascertain any direct correlation [3]. This observation could imply that more complex operations demand both increased time and memory resources, which may inform future optimizations by enabling developers to identify bottlenecks related to resource allocation.

The stress test results underscore the effectiveness of the virtual machine's serialization and deserialization mechanisms, particularly in scenarios involving reconfiguration and data persistence. Serialization demonstrates remarkable consistency in speed, making it highly suitable for preserving data states without significantly impacting performance. Deserialization, while more resource-intensive due to its complexity, remains efficient enough to support dynamic updates with minimal delay. Although outliers in both processes occasionally demand higher resources, these instances are sufficiently rare to allow targeted optimizations. Future work may focus on investigating the correlation between time and memory usage, as well as enhancing the performance of outlier cases to further improve the virtual machine's responsiveness and resource management. Overall, the results affirm that the virtual machine is well-equipped to handle real-time updates and complex configurations, reinforcing its suitability for high-performance, adaptive environments.

4.4. Comparison of Serial, OTA, and Dynamic (VM) Software Updates

In an analysis of firmware update performance for an ESP8266 system, testing was conducted on OTA (Over-the-Air) updates, serial updates, and dynamic configuration updates within a virtual machine (VM). The tests covered various firmware sizes, beginning at 75 KB and progressing up to approximately 1 MB. For the first two types, a basic "Hello World" firmware served as the foundation, with the file size increased by padding it with a byte array. In the VM-based updates, additional size was achieved by embedding dead code or no-operation instructions. The network speed was intentionally restricted to 1 MBps to simulate realistic conditions, while the serial interface was limited to a 9600 bps baud rate, reflecting a standard low-speed connection. The main metric used is the total update time, which spans from the initiation of the update request to the full firmware initialization, with system reset duration—which includes reloading essential modules—and reestablishing network connectivity also being taken into account, applicable only to the OTA and serial updates [29]. This measurement aims to provide a comprehensive overview of the firmware update process under typical constraints.

The results, depicted in Figure 22, reveal a distinct disparity between the different update methods as firmware size increases. The serial interface, due to its limited baud rate, demonstrated the highest update times, which escalated considerably as the firmware size grew. For example, at a firmware size of approximately 900 KB, serial updates required nearly 120 s to complete, highlighting the severe limitations imposed by low baud rates in serial communication. This outcome is expected, as the serial communication speed inherently restricts data transfer rates, particularly when dealing with larger firmware files. OTA updates performed more efficiently than the serial updates, maintaining a more stable update time across the tested firmware sizes. However, OTA updates incurred additional delays due to network variability and reconnection delays. Each OTA update required a network reconnection and a soft reset after the transfer, adding an average delay of several seconds, likely due to the overhead involved in establishing and securing the Wi-Fi connection for each update cycle. Nonetheless, OTA updates demonstrated consistent performance, with fluctuations largely attributable to network factors rather than the update mechanism itself.

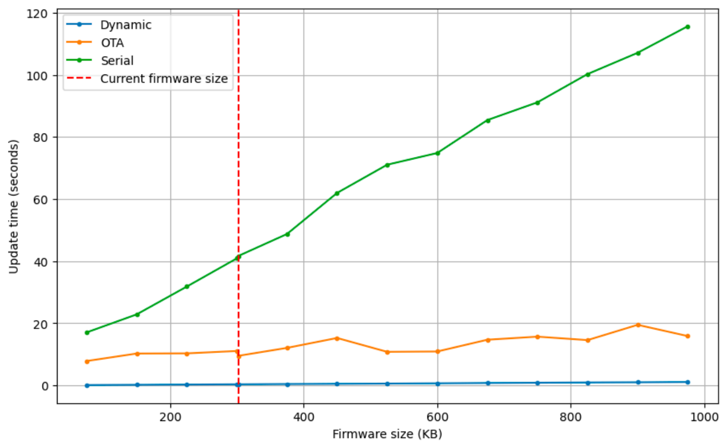


Figure 22. Comparison of update times for dynamic, OTA, and serial firmware updates (authors' contribution).

The dynamic VM configuration updates, which relied on deserialization and dynamic code reloading within the VM environment, displayed a significantly shorter update time. The VM configuration changes consistently occurred in less than two seconds, regardless of the firmware size, demonstrating the efficiency of in-memory updates with minimum reliance on network or serial interfaces. In cases where the configuration reached a size of approximately 900 KB, the VM-based updates averaged around 1.17 s. Of this time, about 1 s was devoted to content transfer, 30 milliseconds to garbage collection, and the remaining time to deserialization and loading the "FunctionObject" into the VM. These results suggest that the bottleneck for VM updates lies primarily in network speed, particularly during the content transfer phase, while the deserialization and initialization steps incur minimal delay, operating in the order of milliseconds. The results presented in Figure 22 underscore the efficiency of VM-based dynamic configuration updates in scenarios where rapid reconfiguration is required. The near-constant update time for the VM configuration, regardless of firmware size, contrasts sharply with the linear increase observed in serial updates and the relatively stable, though slower, performance of OTA. This finding implies that VM-based updates may be particularly advantageous in applications where real-time responsiveness is critical, as they avoid the substantial overhead associated with traditional OTA or serial firmware updates.

The comparative analysis of firmware update mechanisms highlights the limitations and advantages of each approach. Serial updates, while straightforward, are severely limited by the baud rate, making them impractical for large firmware files in time-sensitive applications. OTA updates offer a feasible alternative with moderate performance, though network-induced delays may reduce predictability in environments with unstable Wi-Fi connectivity. In contrast, VM-based updates demonstrate high efficiency and low latency, offering a robust solution for scenarios requiring rapid reconfiguration and minimal downtime. Consequently, in applications where frequent or real-time updates are essential, the VM-based approach appears optimal, leveraging in-memory operations and minimizing reliance on external transfer speeds [29]. Future research could explore optimizing the deserialization and garbage collection processes further to reduce the already minimal update latency in VM-based configurations, potentially enabling even faster response times in dynamic system environments.

5. Concluding Remarks and Future Work

The research and implementation presented highlight the benefits and challenges associated with using virtual machines (VMs) and interpreters for embedded systems, particularly in scenarios requiring flexibility, platform independence, and efficient resource management. By employing a VM layer with an interpreter, this solution enables dynamic reconfiguration and adaptation to changing requirements without requiring extensive hardware modifications or a full system reboot. This capability is especially valuable in applications such as industrial automation, robotics, and autonomous vehicles, where frequent updates, safety-critical real-time responsiveness, and efficient resource utilization are essential.

The stress testing and performance analysis demonstrate that the virtual machine's serialization and deserialization mechanisms perform reliably, with serialization times remaining relatively consistent even under high workloads. Deserialization, while more resource-intensive due to the necessity of allocating objects and handling potential increases in dynamic array sizes, remains within an acceptable range, thus supporting real-time configuration changes with minimal impact on system performance. The garbage collection mechanism is efficient in managing memory without significant interruptions, contributing to the overall stability of the VM. The comparative analysis of firmware update methods, including serial, OTA, and VM-based updates, further illustrates the advantages and limitations of each approach. Serial updates, though straightforward and reliable, are inherently constrained by low baud rates, making them unsuitable for large firmware files in time-sensitive environments. OTA updates offer a viable alternative, although network reliability can introduce variability in update times. The VM-based approach proves to be highly efficient and consistent, achieving rapid updates due to in-memory operations and minimal reliance on external transfer speeds. These results indicate that VM-based updates are preferable for applications requiring frequent reconfiguration or high uptime, as they offer the lowest latency and are less susceptible to network instability.

Despite these promising results, several challenges and areas for improvement remain. One such challenge is the occasional outliers observed in deserialization times, where certain configurations may demand more memory or processing power, leading to temporary spikes in execution time. Addressing these outliers through targeted optimizations, such as refining the garbage collection algorithm or implementing adaptive memory management techniques, could further enhance system stability and predictability, especially for applications with stringent real-time requirements. Additionally, exploring more sophisticated serialization and deserialization algorithms that dynamically adjust to varying data structures could help mitigate resource spikes, ensuring consistent performance across diverse workloads. Future work could also focus on optimizing the network stack and the OTA update protocol to improve reliability and minimize delay variability, particularly in Wi-Fi environments prone to interference or signal loss. Implementing a more resilient networking protocol or incorporating redundancy in data transfer could ensure that OTA updates are more predictable and reliable. Another potential avenue for improvement is enhancing the interpreter to support more complex scripting and logic directly within the VM, allowing for even more responsive and adaptive systems that can process advanced instructions in real time without external intervention [13].

The use cases for this VM-based architecture are extensive. In industrial automation, it allows systems to reconfigure themselves dynamically in response to changing operational requirements, enhancing flexibility and reducing downtime. In autonomous vehicles, rapid firmware updates are essential for maintaining safety and adapting to environmental changes. Furthermore, in IoT networks, where heterogeneous devices often operate on different hardware platforms, the VM's platform-independent layer enables seamless code deployment across various devices, promoting interoperability and reducing maintenance overhead [13]. Overall, the integration of virtual machines and interpreters in embedded systems presents a powerful framework for achieving high performance, adaptability, and low latency in dynamic and resource-constrained environments. Although some challenges remain, the results demonstrate that this approach is well suited to applications requiring real-time updates, platform flexibility, and efficient resource utilization. Future enhancements in serialization, network handling, and garbage collection will further solidify the virtual machine's role as a robust and versatile solution in embedded systems, making it a valuable tool for next-generation applications where responsiveness and adaptability are paramount [4].

Author Contributions: Conceptualization, E.M., E.K. and S.L.; methodology, E.M., E.K., N.Ž. and S.L.; software, E.M.; validation, E.M., E.K., N.Ž., N.B. and S.L.; formal analysis, E.M., E.K., N.Ž., N.B. and S.L.; investigation, E.M., E.K. and N.Ž.; resources, E.M., E.K., N.B. and S.L.; data curation, E.M. and E.K.; writing—original draft preparation, E.M. and E.K.; writing—review and editing, E.M., E.K., N.B. and S.L.; visualization, E.M., E.K. and N.Ž.; supervision, N.Ž., N.B. and S.L.; project administration, E.M., E.K. and S.L.; funding acquisition, E.K. and N.Ž. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: The data presented in this study are available on request from the corresponding author (the data are not publicly available due to further ongoing closed research).

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Wan, C.; Dai, H. Research on Embedded File Management System Based on Forth Virtual Machine. *J. Phys. Conf. Ser.* **2020**, *1486*, 072045. [CrossRef]
2. Peng, L.; Xu, H.; Yu, J.; Liu, X.; Guan, F. EmSBoTScript: A Tiny Virtual Machine-Based Embedded Software Framework. In Proceedings of the 2021 5th International Conference on Computer Science and Artificial Intelligence (CSAI 2021), Beijing, China, 4–6 December 2021; ACM: New York, NY, USA, 2021; pp. 284–289.
3. Elsedfy, M.O.; Murtada, W.A.; Abdulqawi, E.F.; Gad-Allah, M. A Real-Time Virtual Machine for Task Placement in Loosely-Coupled Computer Systems. *Heliyon* **2019**, *5*, e01998. [CrossRef] [PubMed]
4. Lauwaerts, T.; Singh, R.G.; Scholliers, C. WARDuino: An Embedded WebAssembly Virtual Machine. *J. Comput. Lang.* **2024**, *79*, 101268. [CrossRef]

5. Mathew, D.; Jose, B.A. Profiling Applications for a Virtual Machine on an Embedded System. In Proceedings of the 2020 Third International Conference on Advances in Electronics, Computers and Communications (ICAECC), Bangalore, India, 20 December 2020; IEEE: Piscataway, NJ, USA, 2020; pp. 1–6.
6. Polakovic, J.; Mazare, S.; Stefani, J.-B.; David, P.-C. Experience with Safe Dynamic Reconfigurations in Component-Based Embedded Systems. In *Lecture Notes in Computer Science*; Springer: Berlin/Heidelberg, Germany, 2007; pp. 242–257.
7. Marevac, E.; Keleštura, M.; Junuzović, A.; Hodžić, M.; Muhić, S. Design and Implementation of a Virtual Machine for a Dynamic Object-Oriented Programming Language. In *Lecture Notes in Networks and Systems*; Springer: Cham, Switzerland, 2022; pp. 764–784.
8. Bapty, T. Uniform Execution Environment for Dynamic Reconfiguration. In Proceedings of the ECBS'99 IEEE Conference and Workshop on Engineering of Computer-Based Systems, Nashville, TN, USA, 7–8 April 1999; IEEE: Piscataway, NJ, USA, 1999.
9. Scott, J.; Bapty, T.; Neema, S. Runtime Environment for Dynamically Reconfigurable Embedded Systems. In Proceedings of the International Conference on Signal Processing Applications and Technology, Orlando, FL, USA, 1–4 November 1999.
10. Rana, V.; Santambrogio, M.; Sciuto, D. Dynamic Reconfigurability in Embedded System Design. In Proceedings of the 2007 IEEE International Symposium on Circuits and Systems (ISCAS), New Orleans, LA, USA, 27–30 May 2007; IEEE: Piscataway, NJ, USA, 2007; pp. 2734–2737.
11. Liu, X.; Liu, S.; Lv, H. A Dynamic Reconfiguration Scheme for Embedded System Based on Multi-Core DSP. *J. Phys. Conf. Ser.* **2021**, *1802*, 042099. [CrossRef]
12. Dunkels, A. *A Low-Overhead Script Language for Tiny Networked Embedded Systems*; Swedish Institute of Computer Science: Stockholm, Sweden, 2006. Available online: https://urn.kb.se/resolve?urn=urn:nbn:se:ri:diva-22035 (accessed on 4 November 2024).
13. Kang, S.; Kim, H. The Study of the Virtual Machine for Space Real-Time Embedded Systems. In Proceedings of the 2014 IEEE Aerospace Conference, Big Sky, MT, USA, 1–8 March 2014; IEEE: Piscataway, NJ, USA, 2014; pp. 1–7.
14. Wang, W.; Mishra, P.; Ranka, S. Dynamic Reconfiguration in Real-Time Systems. In *Embedded Systems*; Springer: New York, NY, USA, 2013.
15. Zerynth Documentation. Available online: https://olddocs.zerynth.com/r2.3.0/official/core.zerynth.stdlib/docs/vm.html (accessed on 4 November 2024).
16. Industria Italiana. IIoT Plug-and-Play with Zerynth: AI for SMEs. Available online: https://www.industriaitaliana.it/iiot-plug-play-zerynth-pmi-tim-intelligenza-artificiale/ (accessed on 4 November 2024).
17. Damien, P. George, Paul Sokolovsky and Contributors. MicroPython Documentation. Available online: https://docs.micropython.org/en/latest/index.html (accessed on 4 November 2024).
18. Gaspar, G.; Fabo, P.; Kuba, M.; Dudak, J.; Nemlaha, E. MicroPython as a Development Platform for IoT Applications. In *Advances in Intelligent Systems and Computing*; Springer: Cham, Switzerland, 2020; pp. 388–394.
19. Ierusalimschy, R.; de Figueiredo, L.H.; Filho, W.C. Lua—An Extensible Extension Language. *Softw. Pract. Exp.* **1996**, *26*, 635–652. [CrossRef]
20. Machado, A.D.; Fröhlich, A.A. *A Lua Virtual Machine for Resource-Constrained Embedded Systems. Laboratory for Software and Hardware Integration*; Federal University of Santa Catarina: Florianópolis, Brazil, 2010.
21. NetBSD. Lua in the NetBSD Kernel. Available online: https://www.netbsd.org/gallery/presentations/mbalmer/fosdem2012/kernel_mode_lua.pdf (accessed on 4 November 2024).
22. Maakbaas. ESP8266 IoT Framework. Available online: https://github.com/maakbaas/esp8266-iot-framework (accessed on 4 November 2024).
23. Maakbaas. ESP8266 IoT Framework. Available online: https://maakbaas.com/esp8266-iot-framework/ (accessed on 4 November 2024).
24. Suraj151. ESP8266 Framework. Available online: https://github.com/Suraj151/esp8266-framework (accessed on 4 November 2024).
25. Chin, S.; Vos, J.; Weaver, J. *The Definitive Guide to Modern Java Clients with JavaFX: Cross-Platform Mobile and Cloud Development*; APress: New York, NY, USA, 2019.
26. Gilliland, M. The Value Added by Machine Learning Approaches in Forecasting. *Int. J. Forecast.* **2020**, *36*, 161–166. [CrossRef]
27. PlatformIO. What Is PlatformIO? Available online: https://docs.platformio.org/en/latest/what-is-platformio.html (accessed on 4 November 2024).
28. Denaro, G.; Mariani, L. Towards Testing and Analysis of Systems that Use Serialization. *Electron. Notes Theor. Comput. Sci.* **2005**, *116*, 171–184. [CrossRef]
29. Tesone, P.; Polito, G.; Bouraqadi, N.; Ducasse, S.; Fabresse, L. Dynamic Software Update from Development to Production. *J. Object Technol.* **2018**, *17*, 1–36. [CrossRef]
30. Espressif Systems. Available online: https://www.espressif.com/en/products/socs (accessed on 4 November 2024).
31. De Figueiredo, L.H.; Ierusalimschy, R.; Celes, W. The Implementation of Lua 5.0. *J. Univers. Comput. Sci.* **2005**, *11*, 1159–1176.

32. Dada's Perl Lab. The Great Win32 Computer Language Shootout. Available online: https://dada.perl.it/shootout/ (accessed on 4 November 2024).
33. Levis, P.; Culler, D. Maté: A tiny virtual machine for sensor networks. In Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'02), San Jose, CA, USA, 5–9 October 2002; ACM: New York, NY, USA, 2002; pp. 85–95.