FEDERAL UNIVERSITY OF PELOTAS
Technology Development Center
Postgraduate Program in Computing

Thesis

Enhancing Embedded Software in the Internet of Things Domain: Exploring JavaScript on Resource-Constrained Devices

Fernando Luis Oliveira

Pelotas, 2023

Fernando Luis Oliveira

Enhancing Embedded Software in the Internet of Things Domain: Exploring JavaScript on Resource-Constrained Devices

Thesis submitted to the Computer Science Program of the Federal University of Pelotas as a partial requirement to obtain the Ph.D. degree in Computer Science.

Advisor: Prof. Dr. Julio Carlos Balzano de Mattos

Pelotas, 2023

Universidade Federal de Pelotas / Sistema de Bibliotecas
Catalogação na Publicação

O48e Oliveira, Fernando Luis de

Enhancing embedded software in the internet of things domain : exploring javascript on resource-constrained devices / Fernando Luis de Oliveira ; Julio Carlos Balzano de Mattos, orientador. — Pelotas, 2023.

131 f.

Tese (Doutorado) — Programa de Pós-Graduação em Computação, Centro de Desenvolvimento Tecnológico, Universidade Federal de Pelotas, 2023.

1. Embedded software. 2. Interpreted language. 3. Internet of things. 4. Javascript. I. Mattos, Julio Carlos Balzano de, orient. II. Título.

CDD : 005

Elaborada por Simone Godinho Maisonave CRB: 10/1733

Fernando Luis Oliveira

Enhancing Embedded Software in the Internet of Things Domain: Exploring JavaScript on Resource-Constrained Devices

Thesis presented as a partial requirement to obtain the Ph.D. degree in Computer Science, Postgraduate Program in Computing, Technology Development Center, Federal University of Pelotas.

Defense Date: May 4, 2023

Examination Board:

Prof. Dr. Julio Carlos Balzano de Mattos (advisor)
Ph.D in Computer Science from the Federal University of Rio Grande do Sul (UFRGS).

Profa. Dra. Lisane Brisolara de Brisolara
Ph.D in Computer Science from the Federal University of Rio Grande do Sul (UFRGS).

Prof. Dr. Marcio Seiji Oyamada
Ph.D in Computer Science from the Federal University of Rio Grande do Sul (UFRGS).

Prof. Dr. Mateus Beck Rutzig
Ph.D in Computer Science from the Federal University of Rio Grande do Sul (UFRGS).

Prof. Dr. Ulisses Brisolara Corrêa
Ph.D in Computer Science from the Federal University of Pelotas (UFPel).

I dedicate this work to my wife Eliana and my children Miguel and Lucas for their support, patience, and encouragement.

ACKNOWLEDGMENT

Reaching the end of a work as complex, profound, and exclusive as a doctoral thesis requires considerable effort, dedication, and emotional and physical preparation from the student. Therefore, the support of other professionals is essential because we cannot accomplish anything alone.

I would like to express my gratitude to my advisor, Dr. Julio Carlos, who provided me with the right guidance to develop this work. I would also like to acknowledge all the professors and administrative staff at UFPel who have contributed in some way to the completion of my Ph.D.

I would also like to extend my heartfelt thanks to my family, especially my wife, Eliana, for her unwavering support, encouragement, and understanding throughout the past four years. I am also grateful to my children, Miguel and Lucas (by the way, Lucas was born during the course of this work). Your support has been instrumental in my success in completing this work.

ABSTRACT

OLIVEIRA, Fernando Luis. **Enhancing Embedded Software in the Internet of Things Domain: Exploring JavaScript on Resource-Constrained Devices**. Advisor: Julio Carlos Balzano de Mattos. 2023. 131 f. Thesis (Doctorate in Computer Science) – Technology Development Center, Federal University of Pelotas, Pelotas, 2023.

Embedded software development for the Internet of Things (IoT) has predominantly centered on compiled programming languages, such as C and C++, with C being the most widely used. However, the C language has downsides, including lack of object orientation, absence of exception handling, no garbage collection, manual memory management, and other aspects that can make software development challenging, considering the increased complexity of the embedded software requirements. In contrast, interpreted languages like Python and JavaScript (JS) have emerged as alternatives to improve the software quality and abstraction level of applications. Although interpreted languages can bring advantages to embedded software, such as flexibility and ease of use, their execution model (interpretation) can demand higher resource consumption, restricting their use in resource-constrained devices. This study investigates methods to enhance the performance of embedded software to reduce resource consumption in the IoT context, focusing on devices with limited resources. In particular, we chose JavaScript as an alternative to C language and performed investigations to enhance its performance. To do that, we begin with a systematic literature review to understand the relationship between JS and IoT. Then, we analyzed the JavaScript language to understand its impact on constrained devices and performed experiments using benchmarks and real-world applications. As a result, we produced a set of guidelines to improve code quality, a tool (JSGuide) to detect code smells, and developed a framework (JSEVAsync) based on asynchronous functions to help developers build better-embedded solutions. Our findings show that using an interpreted language in embedded software development is feasible and improves design-time metrics, such as maintainability, readability, and code reuse.

Keywords: Embedded Software. Interpreted Language. Internet of Things. JavaScript.

RESUMO

OLIVEIRA, Fernando Luis. **Enhancing Embedded Software in the Internet of Things Domain: Exploring JavaScript on Resource-Constrained Devices**. Advisor: Julio Carlos Balzano de Mattos. 2023. 131 f. Tese (Doutorado em Ciência da Computação) – Technology Development Center, Federal University of Pelotas, Pelotas, 2023.

O desenvolvimento de software embarcado para a Internet das Coisas (IoT) tem se concentrado predominantemente em linguagens de programação compiladas, como C e C++, sendo C a mais utilizada. No entanto, a linguagem C tem desvantagens, incluindo falta de orientação a objetos, ausência de tratamento de exceções, sem mecanismo automatizado para alocal e liberar memória (garbage colletor), gerenciamento manual de memória e outros aspectos que podem tornar o desenvolvimento de software desafiador, considerando o aumento da complexidade dos requisitos de software embarcado. Em contrapartida, linguagens interpretadas como Python e JavaScript (JS) surgem como alternativas para melhorar a qualidade do software e o nível de abstração das aplicações. Embora linguagens interpretadas possam trazer vantagens para software embarcado, como flexibilidade e facilidade de uso, seu modelo de execução (interpretação) pode demandar maior consumo de recursos, restringindo seu uso em dispositivos com recursos limitados. Este estudo investiga métodos para melhorar o desempenho do software embarcado para reduzir o consumo de recursos no contexto IoT, com foco em dispositivos com recursos limitados. Em particular, escolhemos o JavaScript como alternativa à linguagem C e realizamos investigações para melhorar seu desempenho. Para fazer isso, começamos com uma revisão sistemática da literatura para entender a relação entre JS e IoT. Em seguida, analisamos a linguagem JavaScript para entender seu impacto em dispositivos restritos e realizamos experimentos usando benchmarks e aplicativos reais. Como resultado, produzimos um conjunto de diretrizes para melhorar a qualidade do código, uma ferramenta (JSGuide) para detecção de code smells e desenvolvemos uma estrutura baseada em funções assíncronas (JSEVAsync) para ajudar os desenvolvedores a criar melhores soluções embarcadas. Nossas descobertas mostram que o uso de uma linguagem interpretada no desenvolvimento de software embarcado é viável e melhora as métricas de tempo de design, como manutenibilidade, legibilidade e reutilização de código.

Palavras-chave: Software Embarcado. Linguagem Interpretada. Internet das Coisas. JavaScript.

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

API	Application Programming Interfaces
AST	Abstract Syntax Tree
BLE	Bluetooth Low Energy
CLBG	Computer Language Benchmarks Game
CPS	Cyber-Physical Systems
CPU	Central Processing Unit
CSS	Cascading Style Sheets
DOM	Document Object Model
EC	Exclusion Criteria
ESx	ECMAScript(verison number)
ET	Event-Triggered
FIFO	First in First out
GB	Gigabyte
GND	Ground
HTML	Hypertext Markup Language
I/O	Input / Output
IC	Inclusion Criteria
ESP-IDF	Espressif IoT Development Framework
IoT	Internet of Things
IT	Information Technology
JS	JavaScript
JSC	JavaScriptCore
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
kB	Kilobyte
LIFO	Last In First Out

MCU	Microcontroller
MDF	Modified
MHz	Megahertz
mJ	Millijoule
ms	Milisecond
ns	Nanosecond
OS	Operation System
PICO	Population, Intervention, Comparison and Outcomes
PLCs	Programmable Logic Controllers
PPK	Power Profile Kit
RAM	Random Access Memory
RF	Radio Frequency
RFID	Radio Frequency Identification
ROM	Read-Only Memory
RQ	Research Question
RTOS	Real Time Operating System
SDK	Software Development Kit
SMU	Source Measure Unit
SoC	System-on-Chip
STD	Standard
TCP/IP	Transmission Control Protocol/Internet Protocol
TLS	Transport Layer Security
TT	Time-Triggered
VIN	Voltage in
VM	Virtual Machine
XS	Extra Small

CONTENTS

1 INTRODUCTION

In recent years, we have witnessed many technological advances, leading us to a world that is becoming increasingly connected and dependent on technology. We are moving towards ubiquitous and pervasive computing, wherein computer systems become seamless and work to anticipate and respond to users' needs (ABDULSATTAR; AL-OMARY, 2020).

Computer systems are part of the daily life of modern society. They help people in their routines, whether through computers, smartphones, or other electronic devices, with processing capacity. When combined with other equipment or as part of a larger and more complex system, such devices can be understood as Embedded Systems (ES) (BRISOLARA; MATTOS, 2009).

Embedded systems are specialized computer systems designed to perform specific tasks. For instance, they can be used to control a vehicle Anti-lock Braking System (ABS) or airplane altitude control. Therefore, they must be efficient and reliable (WOLF, 2017). Nevertheless, embedded systems can also be used for standard and routine tasks, whether to automate a task or monitor an environment. Therefore, it is increasingly common to find these systems in objects that make up our daily lives, and this combination of computing and everyday objects can be referred to as the Internet of Things (IoT).

The Internet of Things arises from the integration between embedded systems and internet connectivity. In this paradigm, ordinary objects become smart with computational and networking capabilities, using sensors and actuators to interact with the environment, fostering new services, products, and opportunities to innovate (GUBBI et al., 2013). Thus, the IoT transforms traditional objects into smart ones by exploring underlying technologies to detect their execution context, analyzing it, and exchanging information to enable a more productive, safe, and comfortable environment (AL-FUQAHA et al., 2015).

The Transforma Insights Institute (2020) argues that there will be 24.1 billion active IoT devices by 2030, with a growth rate of 11% per year. In a similar positive forecast, IoT Analytics (2023) predicts a 19% growth in the global IoT market size in 2023,

despite the economic downturn. Figures 1 show the forecasts of the IoT market.

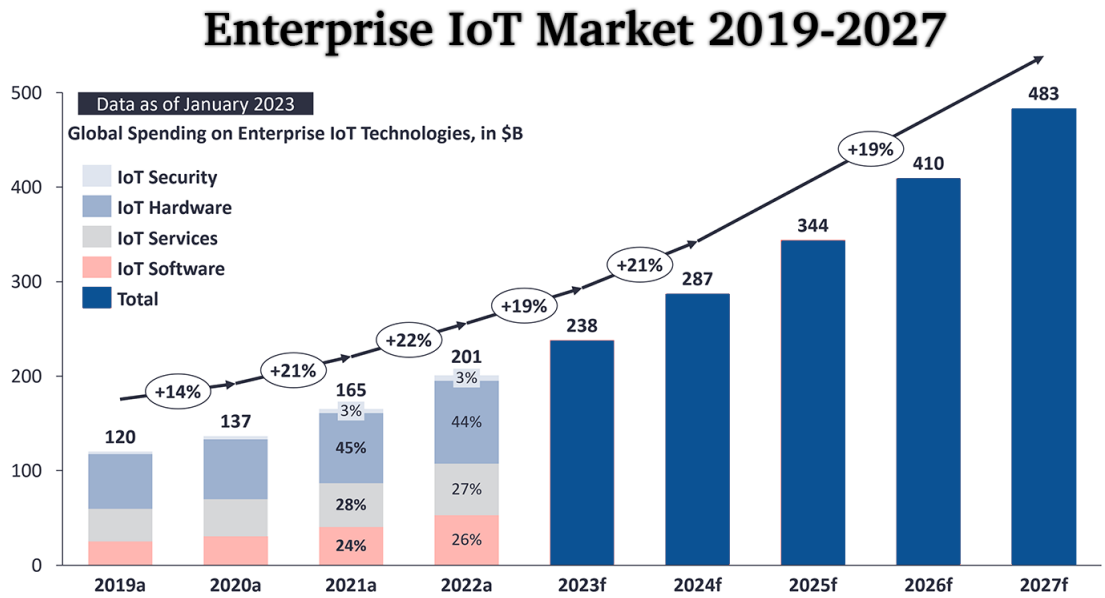


Figure 1 – IoT Analytic market forecast (IOT ANALYTICS, 2023).

The perspective of the Transforma Insights Institute and IoT Analytics confirms the wide range of applications for IoT devices. However, this diversity of equipment presents several challenges in the embedded domain, including reliability, security, storage, communication, and software development.

Software is one of the most critical components of an IoT device. It controls the hardware and applies the functional requirements (NAKAGAWA et al., 2022), where programming languages play an essential role in building embedded applications.

Choosing a programming language is a complex and not trivial decision that software engineers must make (BHATTACHARYA; NEAMTIU, 2011). Several factors might impact the decision-making process, such as implementation cost, quality of the result, learning curve, maintainability, and the language ecosystem. Therefore, each language has different characteristics that should be taken into account when selecting a suitable programming language (FARSHIDI; JANSEN; DELDAR, 2021).

When we bring this discussion to the embedded domain, most designers choose compiled programming languages over interpreted ones (AZIZ; ULLAH; RASHID, 2021). The authors argue that this decision is made because IoT devices are commonly constrained in terms of processing, memory, storage, and power consumption. These characteristics require a software development process that takes into account these resource limitations, forcing developers to consider several issues to meet application requirements and environment constraints (THOLE; RAMU, 2020).

Historically, embedded software development has focused on compiled programming languages like C and C++, with C being the most widely used (SEVERIN; CULIC; RADOVICI, 2020; ECLIPSE FOUNDATION, 2020). However, the C language has

downsides, including a lack of object orientation, absence of exception handling, no garbage collection, manual memory management, and other aspects that can make software development more challenging (PAPADOPOULOS et al., 2018). In contrast, interpreted languages like Java, Python and JavaScript (JS) have emerged as alternatives to improve software quality and the abstraction level of applications.

Interpreted languages are considered easy to learn and use, making them accessible to beginners and new developers. Moreover, they can provide rapid development and prototyping, interactive debugging, and flexibility, which make them platform-independent, allowing developers to write code once and run it on multiple platforms (OLIPHANT, 2007).

While interpreted languages can bring some advantages to embedded software, e, their execution model (interpretation) can demand higher resource consumption, restricting their use in resource-constrained devices (LUBBERS; KOOPMAN; PLASMEIJER, 2022). Therefore, adopting strategies or tools to save resources and improve embedded software became essential.

In this context, there are open opportunities to foster the use and improvement of the execution of interpreted languages for developing embedded software. Thus, this work investigates ways to improve the performance of embedded software by reducing resource consumption and enhancing source-code quality.

In this thesis, we use JavaScript as a programming language to code resource-constrained devices since JS supports multiple programming paradigms, including the event-driven, which is strongly related to the IoT context. Moreover, we have conducted investigations to find ways to enhance the performance of the algorithms. As a result, we produced a set of guidelines and a framework to aid developers in building embedded software for the IoT domain.

1.1 Motivation

The search for improvements in application development for Internet of Things has been investigated in detail by researchers (PRABHU; KAPIL; LAKSHMAIAH, 2018; IWATA et al., 2019; TORRES et al., 2020; HONG; SHIN, 2020; JUNG et al., 2021; KIRCHHOF et al., 2022; WANG; YEN; CHENG, 2023). Many techniques, approaches, and methodologies contribute to better software development. However, the use of interpreted languages in IoT devices, particularly those with limited resources, presents ongoing challenges. Therefore, further investigations addressing these issues are required.

Embedded software development is more challenging than enterprise development because environments traits like limited resources (processing, memory, storage), long operation time, and battery-powered, to name but a few, requiring that the developers

consider these limitations on software development (AZIZ; ULLAH; RASHID, 2021). Thus, these issues can lead to building specific hardware platform-based solutions.

Moreover, embedded software designers have been facing difficulties in designing solutions for modern applications due to the increased complexity of software requirements (LUBBERS; KOOPMAN; PLASMEIJER, 2022). For instance, according to the application domain, the embedded solutions must meet different needs, mixed-critically and safely on real-time application or energy for the battery-based systems. Thus, software development became challenging and the programming language assumes a fundamental role in this process.

The programming language is so relevant that different studies have proposed Domain-Specific Languages (DSL) for the IoT context Eterovic et al. (2015); Koopman; Lubbers; Plasmeijer (2018); González garcía; Zhao; García-díaz (2019); Cacciagrano; Culmone (2020). While these studies can achieve satisfactory results, they are designed to consider specific goals and are limited to certain situations (SALMAN; AL-JAWAD; TAMEEMI, 2021). Furthermore, a significant number of researchers Pattanayak; Patra; Puthal (2013); Salihbegovic et al. (2015); Valsamakis; Savidis (2018); Fabian; Maurice; Christian (2019); Morales; Saborido; Guéhéneuc (2021); Hirasawa et al. (2022) have focused on JavaScript as the basis for their strategies, highlighting the potential and relevance of JS within the developer community.

JavaScript is a high-level, dynamic, and untyped scripting programming language that supports event-driven architecture (FLANAGAN, 2020a). According to Stack Overflow (2022), for ten years in a row, JavaScript has maintained it as the most commonly used programming language.

The JavaScript has become an attractive programming language in the IoT field because it supports event-driven programming, and the JS runtime environment promotes non-blocking concepts through asynchronous functions. This allows for more efficient handling of events and I/O operations (JUNG et al., 2021).

However, the JavaScript language by itself is not enough to guarantee the balance between performance and design-time metrics. Nonetheless, the way the software is developed directly impacts the consumption of resources, being decisive in the success or failure of projects (NAHAS; NAHHAS, 2012).

Application performance is strongly related to code quality (PAPADOPOULOS et al., 2018), but sometimes the abundance of hardware resources can disguise bad coding practices. For instance, if an application runs on a platform with an abundance of resources, hardware can offset the pitfalls of JavaScript (LÓKI.; GÁL., 2018). Thereby, poor coding is more evident on constrained devices and demands attention.

Programming languages enable developers to create different solutions using language resources, including a specific set of functions, utilities, and statements described in the language Application Programming Interface (API) (NASCIMENTO et al.,

2020). Thus, algorithms need to be reviewed since part of the performance issues comes from code that could be written more efficiently (SELAKOVIC; PRADEL, 2016). Furthermore, except for a few cases, developers know little about the impacts of API use because JavaScript engine improvements ensure satisfactory results. Hence, performance regarding coding is a little explored.

In the context of constrained devices powered by batteries, even tiny improvements can mean hours of extended lifetime. This is particularly important for interpreted languages that already require more resources (LÓKI.; GÁL., 2018). Thus, investigating approaches to improve application performance became vital to maintain interpreted language competitive to the compiled ones.

1.2 Hypotheses and Research Questions

Typically, in the context of embedded systems, compiled languages perform better than interpreted languages, leading designers to choose compiled languages and overlook interpreted ones. However, the choice of a programming language should not be based only on performance issues. Other factors, such as time-to-market, maintenance, and code reuse, must also be considered.

This research work has sought solutions for using interpreted languages with high levels of abstraction in embedded devices. We propose the hypothesis that even though an interpreted language, which was not initially designed for embedded software development, might be used as a suitable alternative for coding resource-constrained devices. Furthermore, through the utilization of suitable techniques and adherence to good programming practices, developers can enhance design-time metrics and minimize the performance disparity between interpreted and compiled languages.

The main goal of this thesis is to explore ways to enhance embedded software for the Internet of Things using the JavaScript language, focusing on resource-constrained devices.

Finally, this work answers the leading question: *Can an interpreted language be used to develop high-quality embedded software for devices with limited resources?*. By answering this research question, we are exploring ways to enable the use of an interpreted language that is not commonly applied in embedded software development, particularly in resource-constrained devices. However, this language has the potential to offer benefits in terms of enhancing the development and maintenance of embedded applications in the Internet of Things context.

1.2.1 Contributions

The main contributions of this thesis are summarized as follows:

- **Programming cost**: We demonstrate how much it costs to program using an interpreted language compared to a compiled language. This knowledge aids the designers in understanding the impacts of using JavaScript language for coding embedded software;
- **JSEvAsync**: We developed a framework to assist developers in designing applications for IoT devices using JavaScript language. Our approach combines the benefits of Time-triggered (TT) and Event-triggered (ET) architectures, using JavaScript's non-blocking concept as a development interface to structure algorithms into asynchronous events;
- **JSGuide**: We developed a tool that generates guidelines for embedded software development using JavaScript. Our investigation focuses on identifying techniques or aspects of the JS language that can improve application performance, memory usage, security, and energy consumption,

1.3 Thesis Organization

This section describes the research organization and structure of the thesis content. We have divided the research according to each contribution. Therefore, we provide an overview of the general research process and specific descriptions for each step.

1.3.1 Research Structure

In this study, we performed exploratory research. Stebbins (2001) argues that exploratory research helps researchers investigate existing problems without conclusive results yet. The author also highlights that this technique allows a deeper understanding of a specific problem to answer related questions. Thus, we segmented the investigation into phases to answer the research question: *Can an interpreted language be used to develop high-quality embedded software for devices with limited resources?*. Figure 2 shows an overview of each phase.

We selected the JavaScript language as a case study to validate interpreted languages as an alternative to code IoT devices. Therefore, the research steps consider activities related to it. In particular, the adopted research flow can aid in understanding the problem, performing experiments, and proposing enhancements. In the following, we discuss each research step in detail.

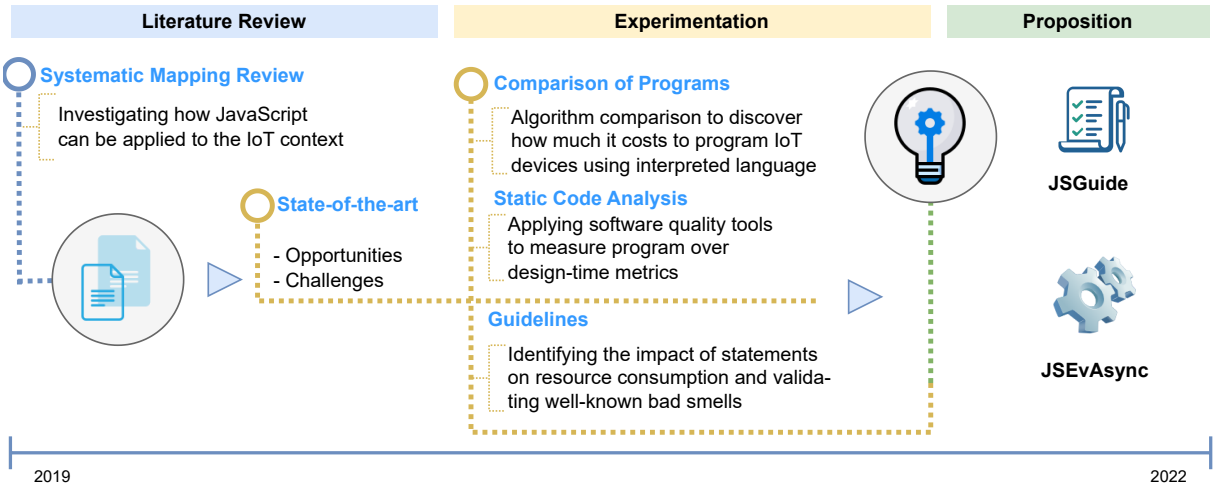


Figure 2 – Thesis organization.

- **Literature review:** This phase aims to investigate the topics of interest for the research to identify opportunities and challenges. We conducted a Systematic Mapping Review to obtain the state-of-the-art regarding using JavaScript in the IoT context; the full literature review can be found in Section 2.4.

- **Experimentation:** This stage represents experiments with technologies or investigations about some approach, method, or technique discovered in the previous phase to improve JavaScript performance. Some studies have helped us to determine which direction to take. For instance, we obtained positive results by applying WebAssembly technology in embedded programming (OLIVEIRA; MATTOS, 2020a). However, considering the typical IoT applications that usually perform sensing, we concluded that it might not be feasible for constrained devices because it can bring a considerable overhead for simple actions; the experiments exploring this issue are discussed in Chapter 3.

- **Proposition:** This phase addresses the design and evaluation of the proposed approach or tool to improve or optimize JavaScript performance in the IoT context. In particular, we focused on understanding the impact of language when applied to resource-constrained devices. Furthermore, we created a framework to aid developers in building more efficient algorithms through asynchronous events. The proposed tools are presented in two chapters: JSGuide in Chapter 4 and JSEVAsync in Chapter 5.

1.3.2 Organization

The results are divided into three segments according to the investigation goal and proposition. Based on the thesis structure, specific tasks were performed to build knowledge about the research topic and translate theoretical knowledge into practical applications. Furthermore, this structured approach ensures that the findings align

closely with the research objectives, enabling a focused and coherent presentation of the results. Figure 3 shows the proposed organization of the results.

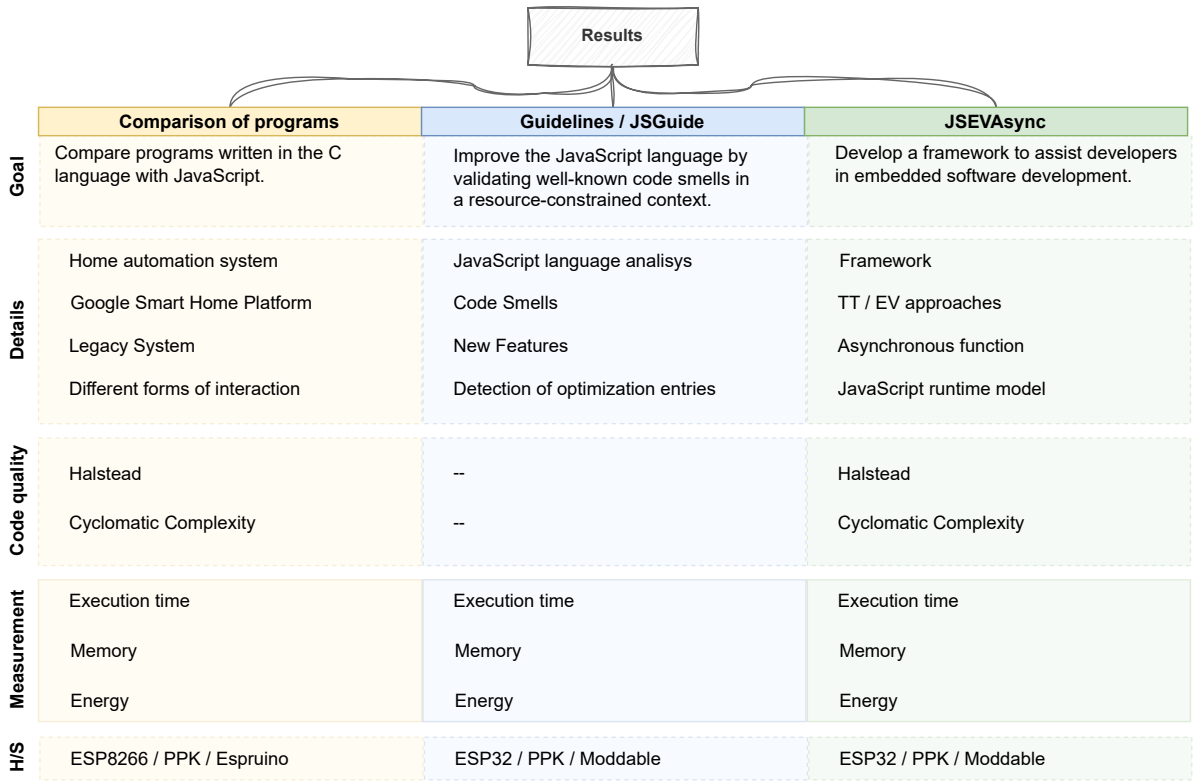


Figure 3 – Results Organization.

Figure 3 represents the segmentation of the results, containing the respective goals, the metrics adopted to measure the data, the resources measured, and the hardware used to perform the research experiment. The first segment explores a comparison between the C and JavaScript languages. The second one explores the JS language based on code smell and the detection of it. Finally, the third segment presents the results based on a new approach to modeling embedded software using asynchronous units based on an event-driven approach.

This structured organization ensures that the findings are presented coherently and logically. Moreover, this allows the reader to easily follow the progression of research and understand how each segment contributes to the overall conclusions. The results regarding the comparison of programs can be found in Section 3.3, while the results related to guidelines and JSGuide are presented in Sections 4.1.2 and 4.2.1, respectively. The results for JSEVAsync can be found in Section 5.2.

The remainder of this thesis is structured as follows: Chapter 2 presents the fundamental concepts related to the research topic and related work. Next, Chapter 3 presents an evaluation of the JavaScript language in the IoT context. In the following, Chapter 4 explores the validation of code smells in embedded software scenarios and also proposes a tool for their detection. Chapter 5 describes a framework designed

to assist developers in improving their algorithms by utilizing an event-driven paradigm with an asynchronous approach. Chapter 6 presents a critical discussion of the research and the achieved results. Finally, Chapter 7 concludes this work and provides an outlook on future research.

1.3.3 Disclaimer / Scope Delimitation

The results presented in this study were obtained through the execution of experiments using the Espruino (ESPRUINO, 2023) and XS (MODDABLE TECH, 2023) JavaScript engines. Most of the results utilize XS because it is the only virtual machine designed for resource-constrained microcontrollers that has nearly full conformance (99%) to the most recent ECMAScript specification.

This thesis did not make any changes to the JavaScript virtual machine. Instead, we restrict ourselves to changing chunks of code, taking into account devices' limited resources to evaluate potential improvements in performance, memory, or energy consumption.

Considering the significant heterogeneity of IoT devices in terms of resources and our focus on resource-constrained devices, we chose microcontrollers with limited capabilities, including only a few kilobytes of RAM/ROM memory and restricted processing capacities, but with networking access, to conduct our experiments. Thus, the results in terms of performance may vary if the experiments reported in this work are reproduced using different JavaScript engines or more powerful microcontrollers. However, the outcomes of the design-time metrics remained unchanged.

2 BACKGROUND AND LITERATURE REVIEW

This chapter provides essential background concepts to help readers better understand the topic at hand. We survey the most directly relevant subjects related to this work, including embedded systems, compiled and interpreted languages, JavaScript, IoT, and a systematic mapping review of the literature.

2.1 Embedded Systems

The miniaturization of computer systems has enabled their integration into machines and objects, allowing for the control and optimization of processes. These integrated systems are commonly referred to as Embedded Systems (ES) (MARWEDEL, 2021).

Embedded systems are designed to serve specific purposes within larger systems. Therefore, they have a wide range of applications, such as cars, trains, airplanes, communication, industry, home automation, to name but a few (WOLF, 2017). Moreover, embedded systems have diverse functionalities and, at the same time, need to be efficient in their tasks. For instance, a smartphone needs to be able to make calls, browse the Internet, take photos, among other functionalities, while maintaining a good cost-benefit relationship between the features performed and battery consumption (BRISOLARA; MATTOS, 2009).

Similarly, Aloseel et al. (2021) defines embedded systems as units that integrate processing capabilities into physical objects to control their functions. Moreover, the success of these systems is mainly due to the inclusion of Programmable Logic Controllers (PLCs), microcontrollers, and microprocessors that enable designers to develop custom solutions similar to traditional IT systems.

Beyond the computer system, and due to its close relationship with the physical process, the integration between computers and the physical world in order to expand their capabilities is known as Cyber-physical Systems (CPS) (BAHETI; GILL, 2011). CPS is not about the union of computers and the physical world. Instead, it is about the intersection of both, regarding the behavior of the cyber and the real world (LEE;

SESHIA, 2017). Thus, CPS represents one segment of the embedded system.

The improvements made to CPS can range from simple issues until to controlling critical tasks. For instance, an airplane's altitude control system needs to be efficient and reliable because a failure in this system can result in physical damage or loss of life. Therefore, these systems are situated on a tenuous line and must be resilient in the face of the challenges posed by context awareness (WOLF, 2018).

Embedded systems face numerous constraints, mainly due to the context in which they are inserted. These limitations are often associated with hardware capabilities that affect the logical components of the solutions, such as programming, data processing, and storage (HEATH, 2002).

Given this context, developing and designing solutions for embedded systems involves physical aspects and, above all, considerations related to the context. Therefore, solutions are increasingly designed to be sensitive and adaptable to the context in which they are inserted (KNAPPMEYER et al., 2013). Moreover, an embedded system has common characteristics that should be considered when building efficient applications to meet software requirements.

2.1.1 Embedded Systems Characteristics

Most users are not aware of embedded systems because they are integrated with equipment, making them invisible. Furthermore, embedded systems are naturally different from each other, but they share common characteristics that distinguish them from traditional computer systems (BARKALOV; TITARENKO; MAZURKIEWICZ, 2019). Moreover, Brisolara; Mattos (2009) argues that projects involving embedded systems must consider several characteristics, which may vary depending on the system's application area. So, some of these traits are highlighted as follows:

- **Reactive system:** In general, most embedded systems are reactive. Through sensors and actuators, these systems monitor and interact with the environment. When a specific event occurs, the system is triggered to perform a particular action or function to handle it;

- **Reliability:** In critical systems, any failure that occurs in an embedded system can result in damage to people's physical integrity, loss of information, or damage to expensive equipment;

- **Real-time:** Some applications have a maximum response time for specific situations; in this sense, embedded systems must respond within the given period. In addition, timing is critical to the correct execution of this kind of system. If it does not meet the deadline, it can lead to failure;

- **Code:** Generally, embedded systems are designed to be integrated into a single

system-on-chip (SoC) with limited processing and storage capacity. Therefore, the source code for the application logic needs to be compact and efficient. Additionally, the development of constrained devices is dominated by the use of the C programming language;

- **Performance:** Embedded systems naturally have limited resources. However, the system must be able to perform its tasks efficiently and within the expected time;
- **Consumption:** In battery-powered devices, one of the main concerns is energy efficiency. An embedded system must be able to perform its tasks while consuming as little energy as possible. However, there is a paradox in which resource consumption conflicts with performance. That is, saving resources can imply not achieving high performance;
- **Size:** Devices are often part of larger, more complex systems. Therefore, factors such as size and weight must be considered as elements in the design of embedded systems;
- **Update:** Some systems do not have mechanisms that allow their program (software/firmware) to be updated remotely, which means that the device needs to have a communication, processing, and storage interface for this purpose;
- **Project time and cost:** Each device has a time to market, which can directly impact the cost and design of the device. If it takes less time to reach the market, the cost of developing the solution will likely be higher.

Beyond these characteristics, embedded systems demand attention to dependability and efficiency requirements. Dependability concerns the physical tasks that the devices perform. For instance, the devices can eventually fail, and therefore, they require repair. Moreover, the equipment should be available as much as possible and operate normally. Efficiency, on the other hand, refers to the optimal use of resources, avoiding waste, and achieving the system's goals (MARWEDEL, 2021).

Dependability is essential for embedded systems because they are often used in critical applications, such as medical equipment, aerospace traffic control, and nuclear power plants. Failures might harm humans and the environment or cause faults in the underlying systems. One way to achieve dependability is by applying fault tolerance techniques (SAHOO et al., 2021).

Fault tolerance is the ability of a system to continue operating despite faults occurring in some of its components (OBERMAISSER, 2005). Therefore, fault tolerance for embedded systems can be achieved through component redundancy or diversity, which involves having more than one item capable of executing the same function.

However, this approach may increase the cost and complexity of the application while reducing the probability of failure. Failures can be caused by hardware failure, environmental factors, power disruptions, and mainly by issues in the embedded software code.

2.1.2 Embedded Software

The logical part of an embedded system comprises the software that controls the device or represents the program's algorithm. Typically, the solutions are static and integrated with the device (SANGIOVANNI-VINCENTELLI; MARTIN, 2001; VANOMMESLAEGHE et al., 2021). Another fundamental aspect of the development of embedded solutions is related to the logical environment of software execution. This includes whether the platform uses an operating system, whether device drivers are supported, and which programming languages are recommended (BRISOLARA; MATTOS, 2009).

In this way, according to Taivalsaari; Mikkonen (2018), devices can be classified into seven categories:

- **Simple device:** Most devices are straightforward. Some equipment such as smart lamps, thermostats, smart sockets, and sensors, in general, do not require complex software stacks. Moreover, they do not need an operating system, and the software is developed specifically for the device;
- **Real Time Operating Systems (RTOS):** This model is similar to a general-purpose operating system. However, in this type of architecture, response time is more important than running several tasks simultaneously;
- **Language-oriented architecture:** Some devices are designed for specific languages or to support virtual machines. Typically, this architecture has more abundant resources such as memory, processing, and storage. For instance, the Espruino[1] and Tessel[2] boards are devices created with support for executing the JavaScript language;
- **Complete operating system:** In this category, devices already have the computing power to support a complete operating system. In addition, the predominant system is based on Linux, and a good example are Raspberry Pi boards (RASPBERRY PI COMPUTERS AND MICROCONTROLLERS, 2023);
- **Application-oriented architecture:** These are devices that can be worn like Android Wear, smart band, and Apple watchOS. These wearable devices have

[1] www.espruino.com
[2] tessel.io

support for third-party libraries, and they are compatible with other devices like smartphones. However, there are minimal hardware restrictions for their execution;

- **Container-oriented architecture:** Usually used in cloud service layers or server processing (back-end), the container is an executable, portable, and autonomous model that allows the isolated execution of applications, facilitating their publication and maintenance.

The first applications for embedded systems used Assembly language, and the goal was to achieve high levels of optimization and performance. However, as the requirements of embedded applications became more complex and dynamic, languages such as C and C++ began to be used in embedded applications (BRISOLARA; MATTOS, 2009).

Although compiled languages are quite efficient and widely used, they require advanced technical knowledge to obtain good performance and generally take longer to develop and maintain. However, they achieve high levels of integration with hardware and perform well (SEVERIN; CULIC; RADOVICI, 2020).

Advancements in hardware technology, such as processing power, storage, and memory, have enabled the use of new programming technologies for embedded systems development, including interpreted languages like Java, Python and JavaScript.

2.1.3 Software Quality Metrics

Software quality metrics are measures used to assess the quality of a software system. These metrics provide objective insights into various aspects of software development, including code complexity, maintainability, reliability, and efficiency (OLIVEIRA et al., 2008a). One of the commonly used software quality metrics is Halstead and Cyclometic complex metrics

Halstead metrics, developed by Maurice Halstead in the 1970s, are a set of software metrics that quantify different characteristics of a program based on its source code. These metrics focus on the size and complexity of the code rather than its functionality. The primary goal of Halstead metrics is to provide a quantitative assessment of software complexity, aiding in predicting potential difficulties and identifying areas for improvement (HALSTEAD, 1977).

Halstead (1977) states that his theory is formed by four principles:

- Program Length (N): It represents the total number of operands and operators in a program. N provides an indication of the program's size, with larger values suggesting increased complexity and potential for errors.

- Program Vocabulary (n): It refers to the total number of unique operands and operators used in a program. A higher program vocabulary indicates a larger set of unique elements, which can impact program comprehension and maintainability.
- Volume (V): Volume is a measure of the program's overall complexity and is calculated using the formula V = N * log2(n). It estimates the effort required to understand and maintain the code. Higher volume values indicate increased complexity and potentially higher maintenance costs.
- Difficulty (D) and Effort (E): Difficulty and Effort are derived from Volume and provide insights into the program's complexity and the effort required for software development. Difficulty (D) measures the cognitive effort needed to understand the code, while Effort (E) estimates the development time required based on D and Volume.

Another important metric is Cyclomatic complexity. Cyclomatic complexity analysis is a software approach developed by Mccabe (1976). It measures the complexity of a program by examining the control flow of the code. The metric is based on the number of independent paths through the source code, indicating the number of unique decision points and potential execution paths.

The control flow graph provides a visual representation of the program's control flow, with nodes representing decision points, loops, and other program constructs, and edges representing the flow of control between these nodes (MCCABE, 1976).

Oliveira et al. (2008a) argues that the use of software quality metrics allows developers to gain insights into the structural complexity of the code and identify areas that may be prone to errors, difficult to understand, or challenging to maintain. Thus, these metrics provide valuable insights for software teams to facilitate refactoring, improve code quality, and deliver reliable and maintainable software solutions.

2.1.4 Compiled and Interpreted Languages

The concepts of interpreter and compiler are inherent to how programs are executed. The application logic or business rules are described using a programming language that can be run through a compiler or an interpreter. However, each programming language adopts an execution model, whether a pre-processed version, compiled or analyzed at runtime (SCHILDT, 1997).

In theory, it is not the programming language that determines how a program will be executed; rather, it is the language designers who make this choice during the language's development. Each language has particularities, advantages, and disadvantages that may make it better suited for one scenario than another (SCOTT, 2000). For instance, the C language may be more suitable for embedded computing if the

objective is performance. However, a high-level language like JavaScript can be better if the focus is on event handling, abstraction, or reuse.

An interpreter is a program that executes instructions written in a high- to low-level language. Two approaches can be used to translate the source code: either by executing it directly and translating it just-in-time or by decoding it into an intermediate representation (bytecode), which is generally more efficient. Moreover, interpreters typically work with programming languages that facilitate the implementation of complex solutions, thereby promoting a high level of abstraction and reducing low-level controls (SCHILDT, 1997). For instance, memory management becomes transparent and actions such as memory allocation and deallocation are no longer the programmer's responsibility. Therefore, developers can focus on building the business logic.

Furthermore, the execution time of an interpreted program is often longer than that of a compiled one. This lag occurs because the interpreter must analyze each line of the program individually and translate it into a language that the processor can understand. In contrast, the compiled code is translated once and runs without overhead. However, for each change in the program, the entire software needs to be compiled again (SCHILDT, 1997; SCOTT, 2000). Figure 4 presents an overview of the execution of the compiled and interpreted program.

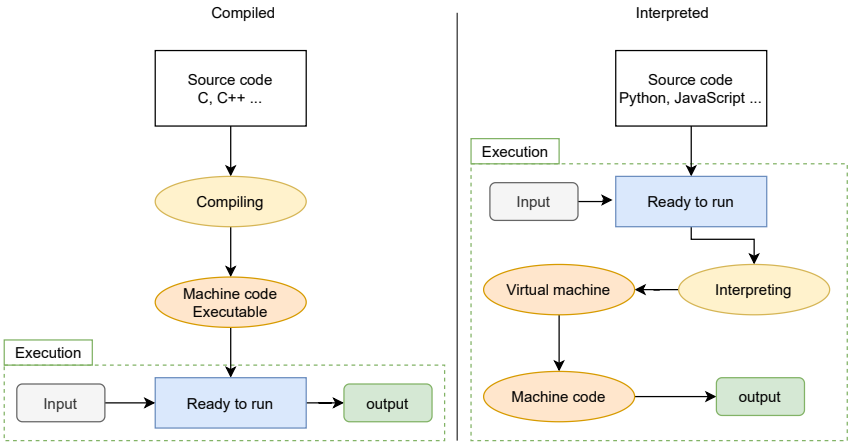


Figure 4 – Compilation and interpretation flow adapted from (SCOTT, 2000).

The Figure 4 shows that compiled solutions are translated into machine language only once, whereas interpreted languages are translated at each execution. During the compilation phase, the compiler analyzes the source code and can introduce optimizations according to the target platform. In contrast, interpreted languages produce a bytecode that is interpreted by the virtual machine (VM) at runtime. The bytecode allows running embedded software on different architectures (ÅSRUD, 2017).

Some languages, such as Java, have a mixed model in which source code is compiled and interpreted. The algorithm is compiled into a bytecode using Java compiler. The bytecode is not a machine code but rather an intermediate representation designed to be executed by the Java Virtual Machine (JVM), which interprets the byte-

code into machine code instructions that are executed on the target environment. This approach is interesting because it allows the same code to be executed on different platforms (GOSLING et al., 2000).

Although interpreted language code goes through more steps, it simplifies programming by allowing developers to write and test solutions quickly. Interpreters can also improve performance by analyzing the source code at runtime to identify frequently used code (hotspots) and perform optimizations. However, these optimizations can increase CPU usage, leading to higher power consumption (SHULL et al., 2019).

Another difference between the language spectrum is regarding debugging. Debugging compiled code can be more challenging because it is translated into machine code, making it more difficult to read and understand. Additionally, any changes to the algorithm require recompiling the entire program, which can be time-consuming. In contrast, debugging interpreted code is usually easier because it is not compiled into a machine code. Consequently, the code can be run through the interpreter stack, allowing developers to identify and fix errors quickly. Furthermore, changes to the code can be made and tested immediately (SCOTT, 2000).

Regardless of the execution model adopted by the language, it is executed or interpreted in the final device. One of the environments that has emerged in recent years and gained attention from industry and researchers is the Internet of Things (IoT).

2.2 Internet of Things (IoT)

According to Ashton et al. (2009) the term Internet of Things (IoT) initially referred to the connection of all physical objects to the Internet, which would have the capacity to capture information through radio frequency identification (RFID) and sensing technologies. Hence, this would allow objects to observe, understand, and interact with the world independently of people.

For Gubbi et al. (2013) IoT is a paradigm in which “objects that surround us will be on the network in one form or another.” These objects or “things” can be devices like lamps, sensors, televisions, and cell phones. These devices are identified and connected to the internet in order to exchange information and make decisions, without human intervention, to achieve common goals.

Similarly, González garcía et al. (2017) claims that IoT is characterized by giving ordinary objects the ability to connect to the network, making them “smart” and capable of capturing, computing, storing, sending, and displaying information. Therefore, objects become able to interact with the environment, generating data in a large quantity, variety, and specificity. From a conceptual point of view, smart objects are based on three pillars related to their ability to (i) be identifiable, (ii) communicate, and (iii) interact with each other (MIORANDI et al., 2012).

In the past few years, the Internet of Things has become increasingly common in many fields of knowledge, driving numerous research projects around the world (PULIAFITO et al., 2019). Terms such as Internet of Services, Internet of Machines, Internet of People, Internet of Knowledge, Internet of Health Things, Social Internet of Things, and Internet of Everything have emerged based on the IoT concept. This technology connects people, machines, and knowledge through the internet, sensors, and actuators. Figure 5 shows an overview of IoT applications and architecture.

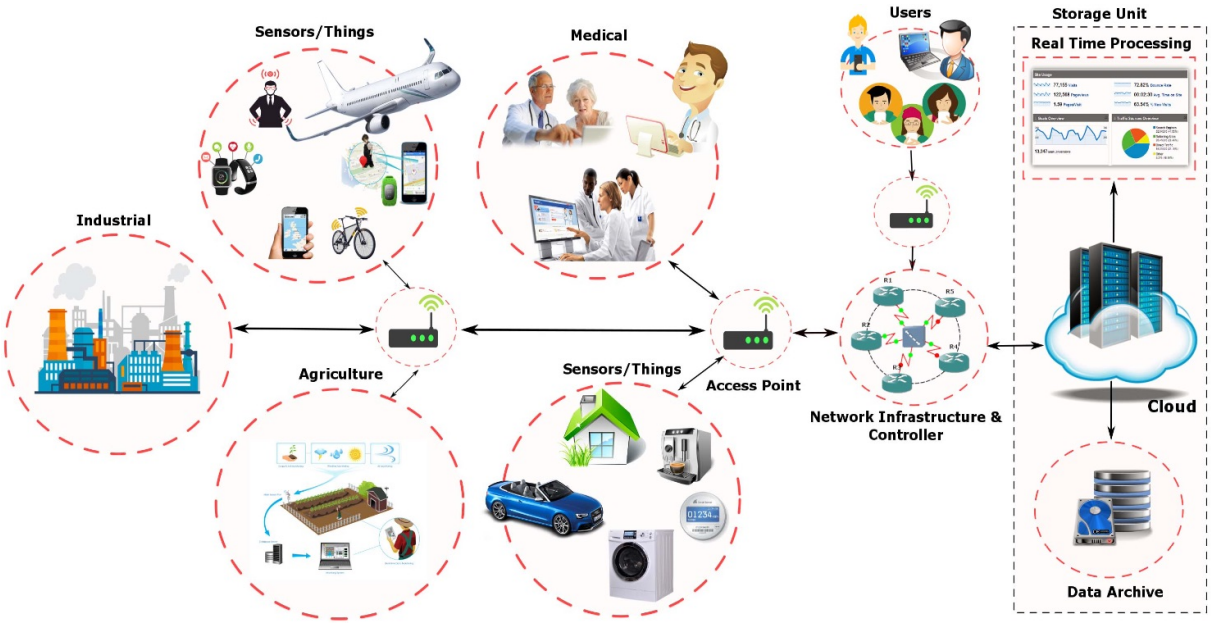


Figure 5 – Overview of IoT architecture (FARHAN et al., 2017).

From Figure 5, it is possible to observe the heterogeneity of IoT devices. In addition, this ecosystem requires support from the underlying layers to transmit, process, and store data. Thus, there are many opportunities to act upon, and challenges to be solved. For example, how can we standardize embedded software coding to reduce time-to-market, decrease costs, and improve design time? Questions such as this still require attention and further investigation.

Given the wide range of actuation of IoT solutions and the diversity of devices, some challenges have emerged. Typically, they are associated with computational, communication, and energy capacities. Therefore, IoT applications must prioritize resource consumption and efficiency (ATZORI; IERA; MORABITO, 2010).

Programming IoT devices needs to be simple in order to achieve success (NAMIOT; SNEPS-SNEPPE, 2014). However, coding means translating multiple rules onto a heterogeneous distributed system. Consequently, programming has been a complex puzzle composed of different types of devices and layers (RILISKIS; HONG; LEVIS, 2015).

In this context, IoT systems work in line with other applications such as servers, mobile clients, or the cloud. Typically, these applications adopt multiple programming

languages, increasing the complexity of the development process since the developer needs to know different languages or requires multidisciplinary teams working together, which can sometimes be challenging to achieve (FARSHIDI; JANSEN; DELDAR, 2021). Nevertheless, to address this challenge, we can adopt a single programming language across all application layers, and for this, JavaScript could be a fitting programming language.

2.3 JavaScript

JavaScript is a scripting programming language whose specifications are provided by Ecma International, a European association that standardizes information and communication systems (ECMA INTERNATIONAL, 2023). In addition, the ECMA-262 group defines the standards and regulations for JavaScript. For this reason, JS is also known as ECMAScript, its official name.

Scripting languages are becoming increasingly popular for developing applications in different areas, with the JavaScript language being a notable example (STACK OVERFLOW, 2022; FARD; MESBAH, 2017). Initially designed for the web, JavaScript has been applied to other contexts and modern web browsers on desktops, game consoles, tablets, and smartphones incorporate JS interpreters, making it a widely used language (CROCKFORD, 2008; FLANAGAN, 2020b).

The success of JavaScript can be attributed to its flexibility and its status as the official language for web applications (WORLD WIDE WEB CONSORTIUM, 2023). As part of the set of standards that define the technology of the World Wide Web, JS enjoys tremendous popularity (GASCON-SAMSON; RAFIUZZAMAN; PATTABIRAMAN, 2017a). A significant factor in its success is the use of efficient virtual machines (VM) to interpret JS code (GAVRIN et al., 2015).

The most popular interpreters for JavaScript are the JavaScriptCore (JSC) engine of WebKit (APPLE JAVASCRIPTCORE, 2023), SpiderMonkey (written by Brendan Eich, the creator of the JS language) (MOZILLA FOUNDATION, 2023), and the V8 engine of Google Chrome (GOOGLE, 2023). In particular, V8 is the core of Node.js, the most popular server-side execution environment for JavaScript (PARK; JUNG; MOON, 2015).

From a technical perspective, JavaScript is a high-level, general-purpose programming language that supports multiple programming paradigms, including functional-oriented, object-oriented, and event-driven paradigms. Also, the JS engine can be implemented as a standard interpreter or just-in-time compiler that compiles JavaScript into a bytecode (FLANAGAN, 2020b). Figure 6 provides an overview of the JavaScript program execution environment.

Figure 6 presents the typical execution flow of a JS program. First, the source code

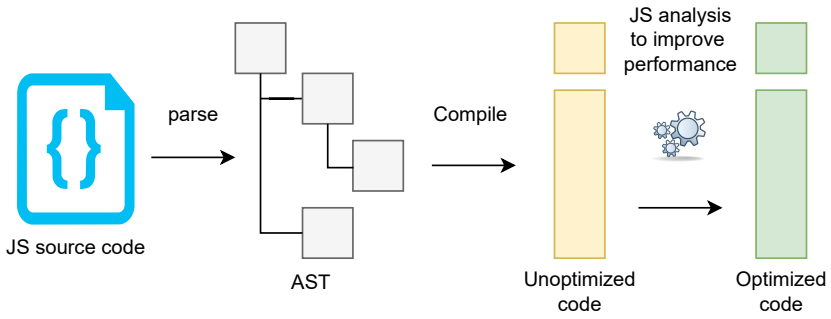


Figure 6 – Overview of the JavaScript execution/optimization pipeline.

must be analyzed to convert the text format into tokens and generate an Abstract Syntax Tree (AST). Then, the engine performs an analysis of the code to check if anything is running slow or if there are bottlenecks or access points that can be optimized. Although this approach is powerful, verifying the code and making decisions about what to optimize requires CPU usage, which can result in higher power consumption. Thus, this extra power consumption can be a significant issue for IoT devices that are typically powered by batteries.

Another relevant trait of JS is its memory model. JavaScript segments memory into two parts: call stack and heap. The call stack area stores primitive values, such as numbers, strings, booleans, or addresses the heap section. In contrast, the heap area stores nonprimitive data, such as objects, functions, and arrays. Figure 7 shows an example of a basic memory model.

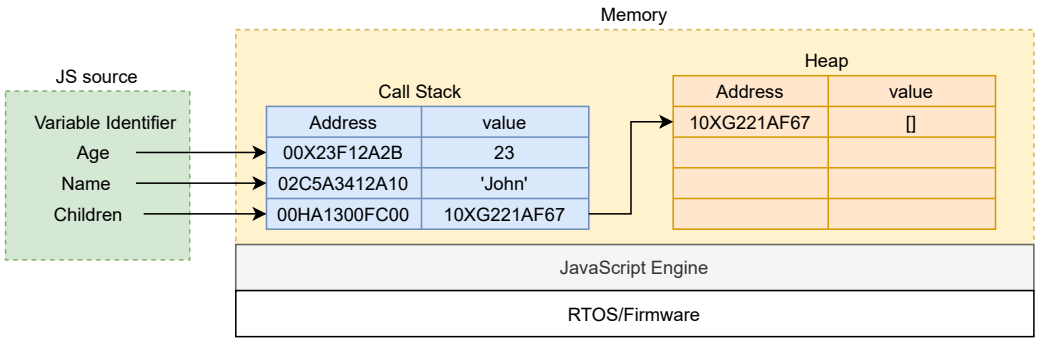


Figure 7 – JavaScript memory model.

The call stack area stores static data that is immutable. Immutable means that data cannot be changed after it has been created, and to modify its content, the creation of a new object with the desired change is required. This strategy allows the engine to determine its size at compilation time; thus, the engine can allocate a fixed amount of space. In contrast, the heap stores mutable objects. The size of an object cannot be determined beforehand; therefore, it occupies more space than necessary because its size is determined at runtime. Therefore, the engine typically allocates memory to the entire heap (ECMA INTERNATIONAL, 2023).

Memory management is a critical issue for system performance, being one of the

main contributors to lower performance and increased power consumption (WOLF; KANDEMIR, 2003). Dynamic allocation can increase peak memory usage and program runtime if it is not efficiently used. Moreover, detecting this type of problem is complex and requires specialized tools for deep profiling (BYMA; LARUS, 2018). Therefore, memory management is a relevant topic that deserves attention from the scientific community to improve this process.

Behind the memory system, JavaScript and its virtual machine solve a traditional problem in computer science, program blocking. Conventional programming languages such as Java, PHP, or C work in a blocking manner by default (NODE.JS, 2023). For instance, if we need to make a network request or read a file, thread execution is blocked until the response is completed. In contrast, the JS strategy uses asynchronous functions to solve the problem using a single thread and event-driven programming. Figure 8 illustrates the JavaScript runtime model.

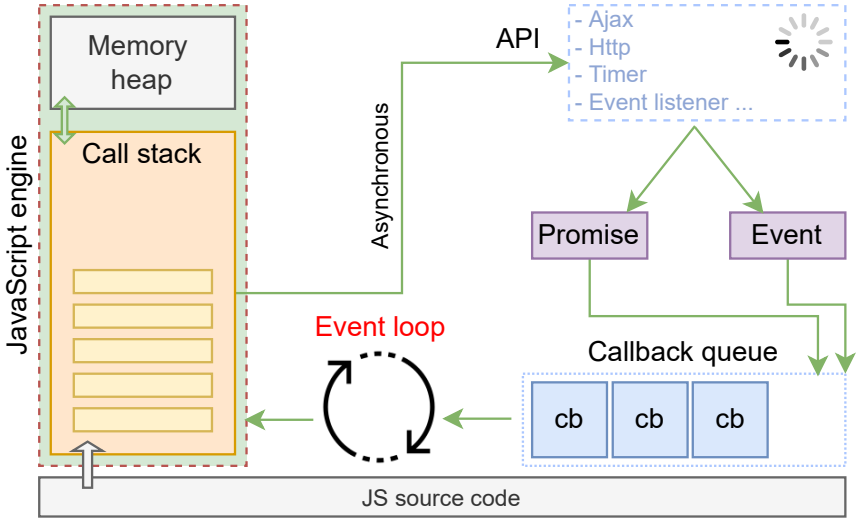


Figure 8 – JavaScript runtime model.

Figure 8 represents the simplest anatomy of a JavaScript runtime environment. First, the source code requires a virtual machine to interpret it. The VM is composed of a call stack and a memory heap. The call stack is responsible for tracking the functions to control what process is running or waiting for any return. In addition, the stack follows the Last in, First Out (LIFO) principle (MOZILLA DEVELOPER NETWORK, 2023). In other words, the first function pushed into the stack will be the last function to be executed and popped off. Thus, the call stack performs the operations in a single thread. Consequently, only one piece of code can be executed at a time, and the data, variables, and other structures surrounding this process (program context) are stored in the memory heap (FLANAGAN, 2020b).

Some operations can be time-consuming and could block the execution process. In this case, asynchronous functions are used to handle them. Asynchronous functions allow operations such as fetching data from a server or getting network status to be

performed asynchronously. In this approach, functions can be registered to handle the response of the operation; for instance, a function (callback) for success, error, or completion status. For that, the JavaScript runtime environment delivers additional features (API) that vary according to the execution context. For instance, Document Object Model (DOM) resources are available in a browser, and sensor/actuator manipulators can be found in an embedded environment (WORLD WIDE WEB CONSORTIUM, 2023).

In general, when an asynchronous operation is initiated, the call stack continues its processing until it completes all functions on the stack. Meanwhile, when an asynchronous operation finishes, the callback associated with it and the operation's results are sent to the Callback queue. Unlike the call stack, the callback queue uses the First-In, First-Out (FIFO) principle and is managed by the event loop (MOZILLA DEVELOPER NETWORK, 2023).

The event loop monitors the callback queue, and when possible, pushes the callback functions to the call stack for processing. A callback is only processed if the call stack is empty, so functions will wait until the call stack has finished processing all the items.

Asynchronous behavior can originate from networking events, digital/analog ports, sensors, actuators, timers, and promises (HODDIE; PRADER, 2020). A Promise represents the eventual completion of an asynchronous (future) operation, and the result is ultimately handled by a callback scheduled by the event loop (FLANAGAN, 2020b).

Although JavaScript is quite popular among developers, its application field focuses on desktops, servers, and mainly on the web. It is a little explored for embedded systems due to the hardware requirements. Therefore, bottlenecks related to performance still need to be investigated to improve JavaScript support, especially for resource-constrained devices.

2.3.1 JavaScript For Embedded Systems

Over time, the interpreted languages have been used in embedded systems context. For example, research such as Clausen et al. (2000), Oliveira et al. (2008a), L; Julian (2015), Pinho; Couto; Oliveira (2019), Han et al. (2019), Mudaliar; Sivakumar (2020) and Cho; Delgado; Choi (2023) explores languages like Java, Rust, JavaScript, and Python from various perspectives as alternatives to the C language. However, these studies have in common the use of equipment with significant computational resources that are capable of running an operating system (OS), such as a Raspberry Pi board (RASPBERRY PI COMPUTERS AND MICROCONTROLLERS, 2023) or even a personal computer.

In contrast, resource-constrained devices comprise equipment with limited computing capabilities, with only a few kilobytes of memory, and cannot support an oper-

ating system. Therefore, the use of interpreted languages in these devices requires specialized analysis and adaptations to optimize resource consumption. In particular, JavaScript language requires a virtual machine design that considers the target environment's limitations.

Beyond being limited in terms of hardware, IoT devices typically operate under an event-driven model in which actions and responses are guided by inputs and outputs (I/O) from sensors, actuators, or networks (KIM; JEONG; MOON, 2017). In other words, IoT devices are event-friendly, and embedded programmers can create countless handlers for events that can be periodically fired by users or through sensors. Therefore, coding programs over an event-driven model is a natural choice for IoT devices.

JavaScript becomes attractive as a programming language in the IoT domain because it supports the event-driven programming model, where functions can be registered as event handlers. In addition, functions are viewed as objects in JavaScript and can be declared anonymously, making it easier to bind them with the event handler (HODDIE; PRADER, 2020).

In addition to supporting the event-driven model, JavaScript brings to the embedded context the possibility of working with non-blocking routines through asynchronous functions. This allows common actions such as reading data from sensors or sending data over a network to be performed asynchronously (JUNG et al., 2021).

From a design-time perspective, JS can enable rapid prototyping and maintainable code, while also increasing the portability of embedded programs (UGAWA; IWASAKI; KATAOKA, 2019). It means that JavaScript applications can be embedded into heterogeneous environments (ANDREASEN et al., 2017).

Regarding security, the JavaScript language provides mechanisms that allow secure coding and execution. For instance, common security bugs in native code, such as uninitialized memory and buffer overruns, do not occur in the JS. Moreover, the language is constantly being updated, with new features released each year to improve security and simplify coding aspects. The specification also has a rigorous and well-structured process to evolve the language (ECMA INTERNATIONAL, 2023).

JavaScript supports Transport Layer Security (TLS) to transfer data from IoT devices to server/cloud systems, ensuring the privacy of generated data and preventing tampering with sensitive user data. In addition, it is possible to establish secure communication with other devices using standard protocols such as ZigBee and Bluetooth Core Specification (BLE). Furthermore, executing scripts in the sandbox model is feasible and allows the creation of specific security policies (FLANAGAN, 2020b).

Finally, the JavaScript language we expose is the same language used for general purposes; it is important to motivate developers to use it. However, JS has also been concerned with embedded systems. The first JavaScript API specification for embed-

ded systems was recently released (ECMA INTERNATIONAL: TECHNICAL COMMITTEE 53 , 2023). The ECMA-419 specification brings JS closer to hardware, promoting standard access. This approach allows for the writing of portable scripts that can be deployed on different devices. Therefore, JS has taken a significant step forward in strengthening its application to embedded software development.

Naturally, the use of JavaScript depends mainly on hardware capabilities. However, the progress of virtual machines allows for the use of modern JS in constrained devices. XS engine (MODDABLE TECH, 2023) is an example of a virtual machine designed for microcontrollers.

2.3.2 Moddable XS JavaScript engine

To support JavaScript in embedded systems, a virtual machine is required to interpret the scripts. Some engines focus on embedded systems such as Espruino (ESPRUINO, 2023), JerryScript (JERRYSCRIPT, 2023), Ducktape (DUKTAPE, 2023) , QuickJS (FABRICE BELLARD, 2023) , and mJS (MONGOOSE OS, 2023); however, they are limited. To keep engines small and compact, they restrict their support to only a subset of the JavaScript programming API. This means that part of the resources of the language cannot be covered, and therefore, are not available for use.

In the opposite direction of other engines, one company developed an engine created exclusively to serve resource-constrained microcontrollers. Moddable Tech Inc. maintains the XS engine, which has some characteristics that make it unique. From this point, all the information related to Moddable SDK and XS engine was provided by the authors themselves and were extracted from their official repository (MODDABLE GITHUB, 2023) and website (MODDABLE TECH, 2023).

First, the XS engine is the only engine that implements more than 99% of the most recent version of the JavaScript language specification (i.e., more than some browsers). Second, Moddable provides a complete toolchain to support all steps of JavaScript development, including tools and runtime software to create IoT applications using standard JavaScript on resource-constrained microcontrollers.

The traditional JavaScript engine for the desktop or server side has a main focus on speed, which conflicts with the IoT domain because the high-speed demands extra resource consumption to achieve a speed-up. In contrast, the XS engine is designed to minimize resource usage, especially RAM. Therefore, XS uses a distinct approach, similar to mobile development, where the application is described using a manifest and the code is compiled into an optimized and small bytecode that is interpreted on the Microcontroller (MCU). Figure 9 presents an overview of the development process using Moddable SDK.

Moddable's philosophy supposes that embedded software development is not a standalone activity. Typically, embedded developers build the application using a per-

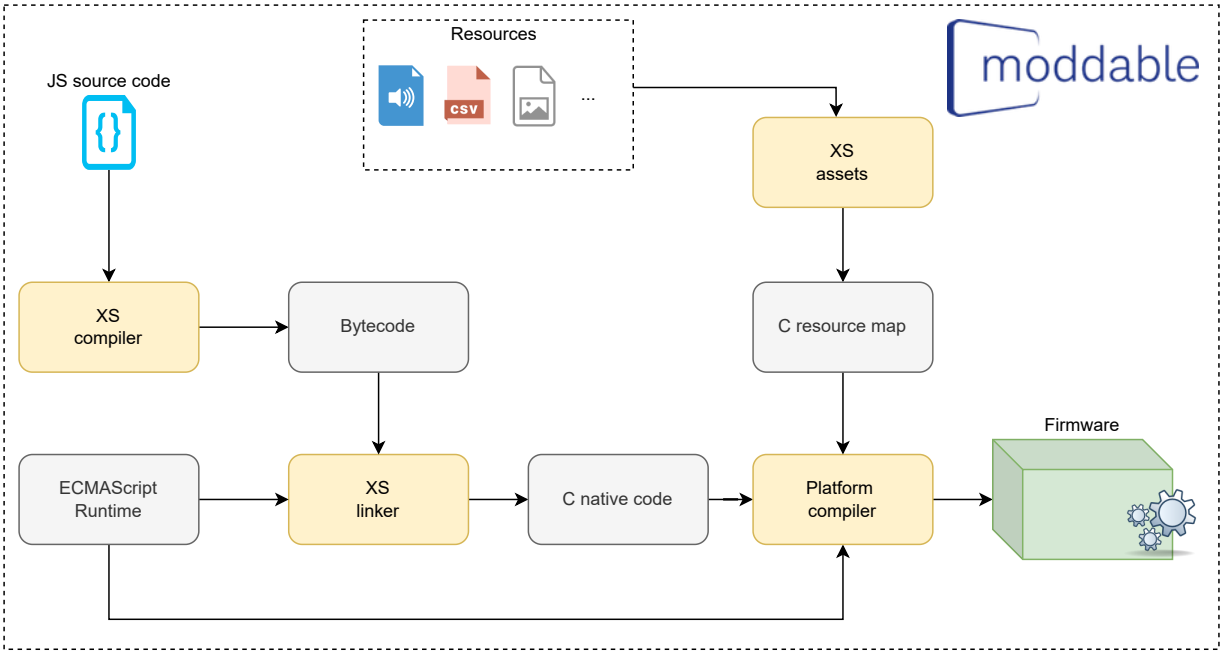


Figure 9 – Moddable approach (MODDABLE TECH, 2023).

sonal computer that has significantly more performance than an IoT device. Therefore, the Moddable toolchain explores the extra power to compile and optimize the bytecode at design time.

First, the XS compiler parses the source code and builds bytecode. Next, the XS linker analyzes the bytecode and pre-executes it, selecting the built-in objects and removing the unused objects to reduce the required ROM space. Also, it is possible for the JavaScript code to call functions written in native code (using C language) targeting performance, and XS handles the calls between languages.

Optionally, if the application uses assets like audio, images, XML, or another kind of file, XS has a specific encoder to map the resources. Finally, the XS platform compiler generates firmware with the application. All of these steps are executed using a command-line tool that facilitates and standardizes the developer process.

Moreover, several improvements are applied during the compilation phase to optimize the algorithm. For instance, all built-in functions, classes, and prototype objects are saved as native data, which is flashed into ROM instead of RAM. Furthermore, to achieve performance and optimize memory usage, the scope of the variables is analyzed, and a syntax tree is produced to assign variables by index instead of having to look up identifiers. As a result, in some cases, we can execute JavaScript with performance closer to the native language.

If performance remains a problem, XS allows the integration of native code into JavaScript projects. The XS interface in C enables the use of C code to optimize time-consuming routines or reuse existing C and C++ libraries to improve the algorithm performance (HODDIE; PRADER, 2020).

2.4 Systematic Mapping Review: JavaScript Applied to IoT

This section describes the literature review regarding the application of JavaScript in the context of the Internet of Things. Our review was performed in two periods. The first one was executed in 2019 and considered papers written from 2009 to 2019 (10 years). The second phase represents the review update that includes the most recent publications on the research topic. Therefore, the final literature review considers research published in the last thirteen years, representing the state of the art in the integration between JS and IoT.

We conducted a Systematic Mapping Review of the literature to get an overview of the research area. To do that, we followed the guidelines proposed by Petersen et al. (2008). Therefore, the leading question that motivated this research was: *How can JavaScript be used in programming for the Internet of Things?*

In order to simplify the investigation, the general question was broken down into a smaller set of research questions (RQs):

- RQ1: What are the purposes of using JavaScript in the IoT?
  - The goal of this question is to discover the motivations for applying JavaScript to IoT.
- RQ2: What are the most investigated research topics, and how have they changed over time?
  - The goal is to group the researchers by areas and analyze how research topics have evolved over time.
- RQ3: Which JavaScript interpreters (virtual machine) are found? What level of JavaScript API is present?
  - The goal is to define which engines are used and which features are available..
- RQ4: What are the biggest challenges in developing IoT applications using JavaScript?
  - The goal is to identify JavaScript's limitations and opportunities for application in IoT.
- RQ5: What tools or plugins are available for JavaScript in the Internet of Things context?
  - The goal is to identify the available resources to support integration among technologies or tools to enhance the development process.

To conduct the research, a set of keywords was generated based on the research questions. To systematize this step, Kitchenham et al. (2010) suggests using the PICO process, which stands for Population, Intervention, Comparison, and Outcomes. Table 1 presents the generated keywords, while Figure 10 shows the basic search string.

Table 1 – Pico model for the framing question.

Criteria	Goal	Words
Population	IoT devices	IoT OR Internet of Things
Intervention	JavaScript Interpreter	JavaScript
Comparison	not applied	
Outcomes	Integration between JavaScript and IoT	JavaScript AND IoT

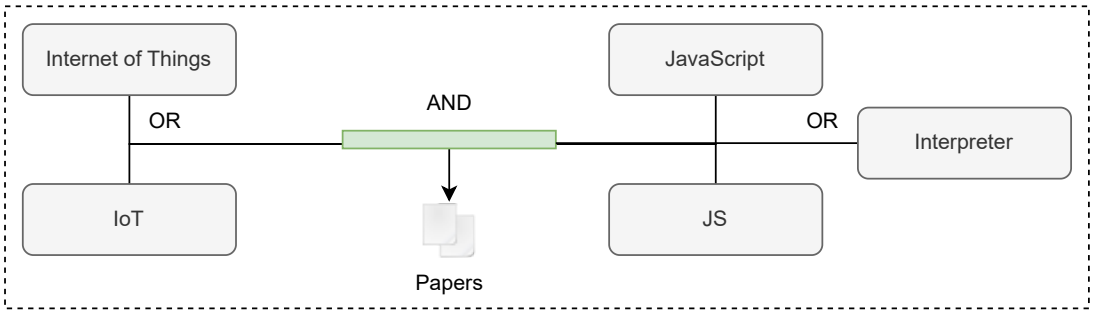


Figure 10 – Basic string search.

Regarding the data sources, we choose the IEEE Explore[3], Elsevier[4] and ACM Digital Library[5] as libraries, and Scopus[6] and Engineering Village[7] as indexers.

As an instrument for selecting or excluding papers, we utilized specific filters. Table 2 shows the criteria that were used, and Figure 11 illustrates the search process.

Table 2 – Inclusion and Exclusion Criteria.

Type	Ref	Description
Inclusion	IC1	Papers written in English or Portuguese
	IC2	Peer-reviewed articles in journals and conferences
	IC3	Published from 2009
	IC4	Papers related to JavaScript language applied to IoT
Exclusion	EC1	Duplicated papers
	EC2	Papers not available for downloading
	EC3	Paper lies outside the IoT and JavaScript language domain
	EC4	Paper written by us

Figure 11 – Papers selection process.

[3]https://ieeexplore.ieee.org
[4]https://www.elsevier.com
[5]https://dl.acm.org
[6]https://www.scopus.com
[7]https://www.engineeringvillage.com

The paper selection process was divided into four phases. The first phase involved executing the research based on the search string. The search process allowed for applying inclusion and exclusion criteria, although this could vary depending on the search interface of each data source. In the second phase, duplicate items were removed from the indexed databases. The indexer typically retrieves papers that have already been searched in the digital libraries. The third phase consisted of reviewing the title and abstract of the papers to select those that align with the research's goals, thus choosing papers for deeper analysis. Finally, the fourth phase involved analyzing the full text. Figure 11 also presents a fifth step, representing data extraction from selected papers. Figure 12 shows the number of papers selected at each step.

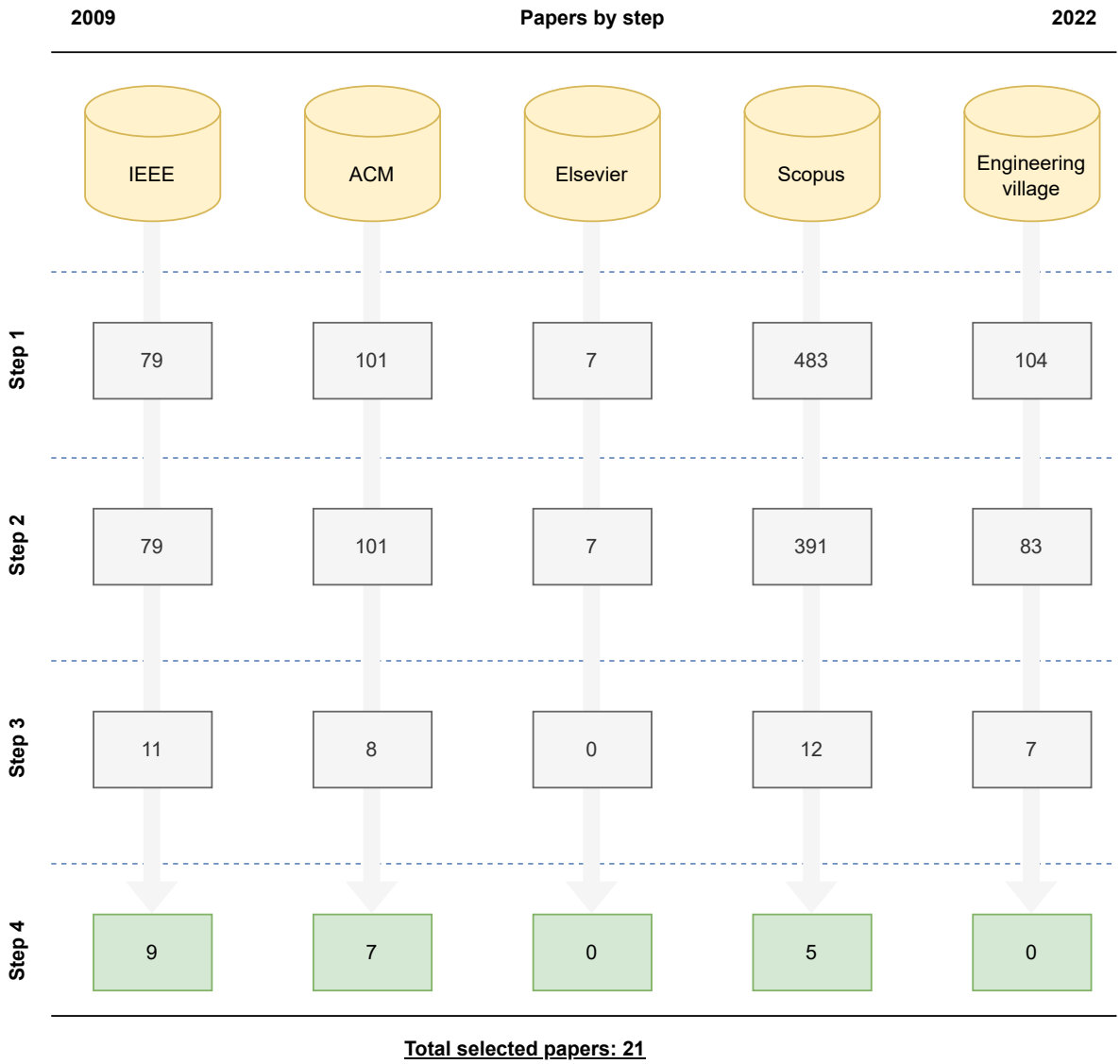


Figure 12 – Papers selected by step.

Based on the 21 selected papers, an analysis was performed to answer each research question. During the data extraction process, the same paper may have been used to answer more than one question. Figure 13 depicts the temporal distribution of the publications.

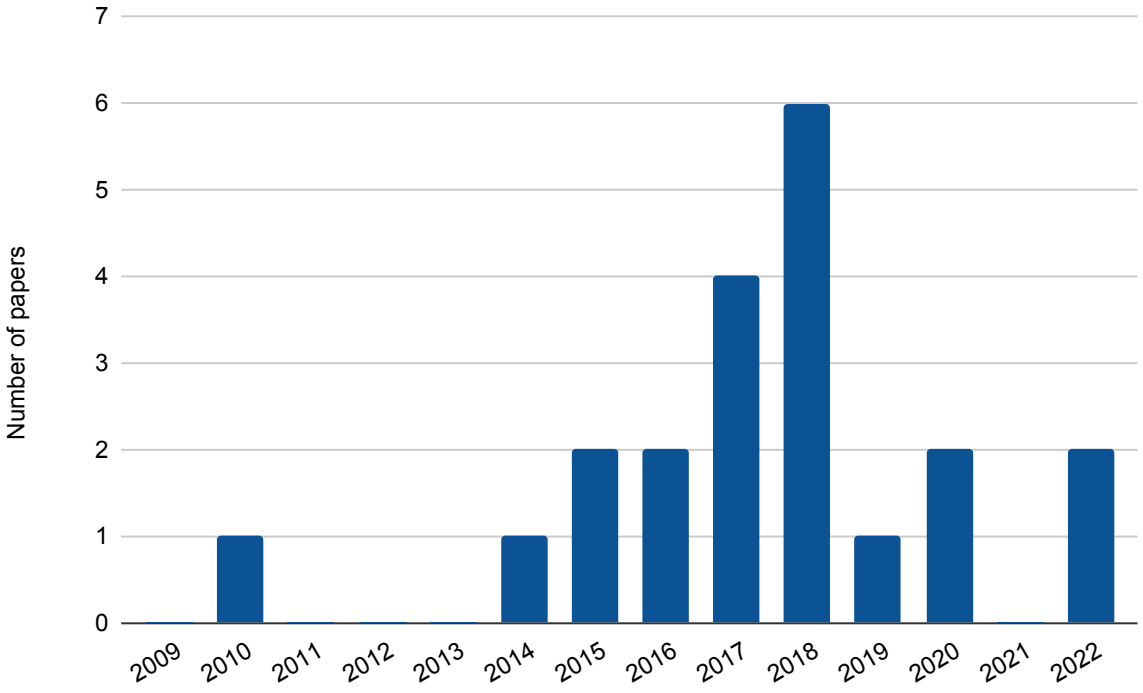


Figure 13 – Amount of papers by year.

RQ1: What are the purposes of using JavaScript in the IoT?

The use of JavaScript in the context of the Internet of Things can be viewed as a natural choice because it enables the event-driven paradigm, which is one of the most commonly used models for IoT (KIM; JEONG; MOON, 2017).

Although it may seem like an empirical analysis, it is possible to verify that adopting JS as a language in IoT development results in an environment communicating through a single programming language. Also, another reason to use JS in the IoT context can be summarized as follows:

- **Ubiquity:** JavaScript is one of the most widely used programming languages in the world, ensuring a robust and active community. This means that much support is available, including libraries and frameworks, which can simplify the development process;
- **Familiarity:** Many developers are already familiar with JavaScript, which makes it easier to start with IoT development. It can also help to reduce the learning curve, as programmers do not need to learn a new language from scratch;
- **Code reuse:** Using the same technology in both application layers (IoT device, server, and client) allows for code reuse, simplifies logic, and reduces the time and cost of development;

- **Teaching programming:** JavaScript can be used to teach programming through physical objects, contextualizing theories and practices;
- **Debugging:** JavaScript, as an Interpreted Language, can favor application debugging because it allows developers to step through the code line by line, and debugging techniques provide valuable insights into code behavior, helping developers identify and fix bugs more efficiently.

One of the activities that we believe will be a trend is contextualized programming teaching. For instance, Peterson; Vogel (2018) used JavaScript in their work to teach programming through physical objects, thereby correlating theories and practices.

Another fact that caught our attention is that some research Gascon-samson; Rafiuzzaman; Pattabiraman (2017b); Gavrin et al. (2015); Li et al. (2018); Heo et al. (2015); Kirchhof et al. (2022); Ugawa; Marr; Jones (2022) investigates virtual machines to improve existing machines or create new ones. It is clear that engines are a hot topic in this context and are critical to expanding JS support on resource-constrained devices.

RQ2: What are the most investigated research topics, and how have they changed over time?

Each paper can cover different topics and have discussions on several subjects. To classify and group them, we assigned each paper to only one area to facilitate the identification of research trends and opportunities. Thus, each paper was framed within a specific topic, as follows:

- **Security:** It fits in this category of papers that address security aspects in the execution of JavaScript;
- **VM:** Comprises papers that propose some virtual machine or improvements in the execution of JavaScript in the IoT context;
- **Migration:** This set of articles deals with migration and stateful challenges, as well as the serialization of information for the JavaScript language that needs to be transferred to another device;
- **IoT/JS:** This category includes papers that discuss both IoT and JavaScript without a specific focus on either topic;
- **Prototyping:** Papers that focus on prototyping IoT solutions considering the JavaScript scripting language to teach programming;
- **Interoperability:** It fits in this category of papers that address interoperability through JavaScript;

- **Interface:** We consider the papers that present some application interfaces as a platform in which JavaScript is used as the basis for constructing IoT solutions. It may include the use of case studies.

Figure 14 shows the generated taxonomy from the research and the number of papers that fit inside each category, while Table 3 lists the included papers.

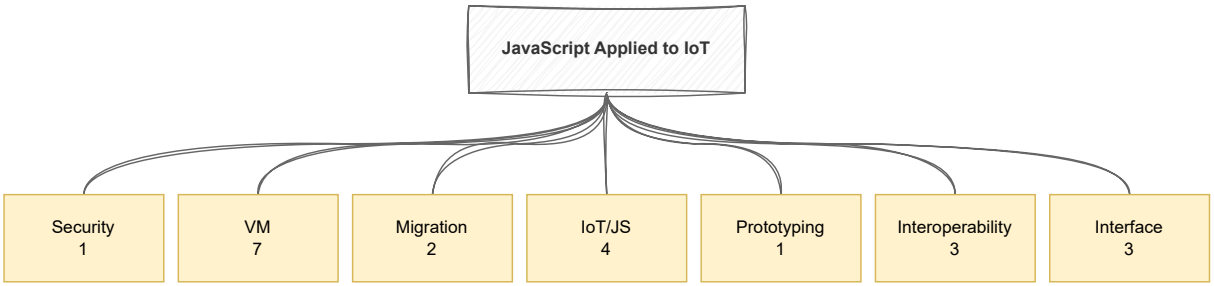


Figure 14 – IoT – JS taxonomy.

Table 3 – Papers by topic.

Topic	Papers
Security	Sahu; Singh (2016)
VM	Gascon-samson; Rafiuzzaman; Pattabiraman (2017b), Gavrin et al. (2015), Li et al. (2018), Heo et al. (2015), Grunert (2020), Morales; Saborido; Guéhéneuc (2020), Ugawa; Marr; Jones (2022)
Migration	Kwon; Moon (2017), Gascon-samson et al. (2018)
IoT/JS	Jaimini; Dhaniwala (2016), Guinard et al. (2010), Baba-cheikh et al. (2020), Wang et al. (2022)
Prototyping	Peterson; Vogel (2018)
Interoperability	Gascon-samson; Rafiuzzaman; Pattabiraman (2017a), Ghosh; Jin; Maheswaran (2014), Gascon-samson; Jung; Pattabiraman (2018)
Interface	Lee et al. (2017), Baccelli et al. (2018), Bak; Chang; Choi (2018)

Regarding the evolution of the research topic over time, it was initially observed that the research did not focus on JS or IoT, as addressed in Guinard et al. (2010)'s study. Instead, such technologies act as underlying support for other projects or applications. However, the concept of intelligent objects has gained prominence, and challenges, such as interoperability, security, and resource consumption, have emerged. Furthermore, the relationship between JS and IoT gained strength in 2015, coinciding with the release of the new JavaScript specification (ES6). We believe that the new version was motivated to expand the language to several areas, including IoT.

RQ3: Which JavaScript interpreters (virtual machine) are found? What level of JavaScript API is present?

This question explores the alternatives for enabling JavaScript execution on embedded systems. Table 4 presents the identified options along with their size characteristics and resource coverage, according to the ECMA-262 specification.

Table 4 – JavaScript engine for IoT.

Engine	Size (kB)	JS Version	Compliance (%)
WebletScript	103	5.1	<60
v7	120	5.1	100
JerryScript	160	5.1	100
Duktape	184	5.1[a,b]	99,4
Espruino	231	5.1[b]	<70
Moddable XS	64 [c]	13	99

[a] Partial/initial support for ECMAScript version 6
[b] Partial/initial support for ECMAScript version 7
[c] The size is defined according to the API used.

In this research, a controversial point was identified when discussing virtual machine for IoT devices. Some devices are limited in technical terms, and as a result, the WebletScript interpreter (LI et al., 2018) chooses not to implement some language specification definitions in order to produce a smaller binary file. On the one hand, reducing the size can enable the interpreter to be used on a higher number of devices. On the other hand, not covering certain technical aspects of the specification may restrict the adoption of the interpreter.

Some language features are not commonly used, but they are still part of the language and should be present. It is a difficult decision to make whether to prioritize a greater number of supported devices and limit the language's capabilities or to implement all the features of the specification and reduce the number of supported devices. One effective strategy to mitigate this problem is to adopt Moddable's strategy, which performs pre-execution of the script to remove unused objects and reduce the footprint size (see Section 2.3.2).

Over time, devices have improved their technical capabilities, and therefore, it is best to implement as much of the language specification as possible. This allows the virtual machine to be compatible with a larger number of libraries and, consequently, gain community confidence and credibility.

RQ4: What are the biggest challenges in developing IoT applications using JavaScript?

The IoT poses many challenges in several dimensions, including hardware heterogeneity, different operating systems, programming languages, APIs, and various communication protocols (LI et al., 2018). Moreover, the lack of a standard implies dealing with distributed and complex systems (GASCON-SAMSON; RAFIUZZAMAN; PATTABIRAMAN, 2017b).

Indeed, the biggest barrier to enabling JavaScript on embedded devices is the computational limitation of these devices. For instance, some devices are already designed to be extremely limited, with resources only sufficient for performing a single activity.

Also, these devices often have size requirements, such as those used in the healthcare area, where they need to be as small as possible. Therefore, certain devices will always have technical limitations that restrict the use of JavaScript as a programming language (TAIVALSAARI; MIKKONEN, 2018; MIRAZ et al., 2015).

Resource consumption is a significant concern when using JavaScript on IoT devices. This issue is not unique to JavaScript and is a recurring problem in many programming languages. However, the fact that JavaScript is an interpreted language can further exacerbate this problem on devices with limited resources. Therefore, it is crucial to consider factors such as energy efficiency, bandwidth, and memory usage when developing IoT solutions using JavaScript (LI et al., 2018; GUBBI et al., 2013).

We identified a promising trend in the computational processing at the IoT edge. The advancement in the processing and storage capabilities of devices now enables local processing of some information on the edge device, rather than sending it to an external agent such as a cloud service. This approach innovates the classic IoT cloud-centric model (GASCON-SAMSON; JUNG; PATTABIRAMAN, 2018; TAIVALSAARI; MIKKONEN, 2017) by avoiding network overhead and reducing the response time because of the local processing of data. Thus, investigations into object state migration support distributed processing in an IoT context.

One of the great challenges of the Internet of Things is to understand and anticipate the wishes of end-users. To achieve this goal, objects need to monitor and observe user movements, gestures, locations, and contexts (MIRAZ et al., 2015). From this data, opportunities for action emerge. Artificial intelligence techniques can be applied to understand human beings, the environment, and interact appropriately.

RQ5: What are the tools or plugins for JavaScript in the Internet of Things context?

By answering this question, we are exposing the technologies and tools identified in this survey. However, we do not address virtual machines because they have already been discussed in Research Question 3. The remaining technologies are as follows:

- SmartJS: JavaScript-based middleware; that provides an environment of execution and development of IoT solutions (GASCON-SAMSON; RAFIUZZAMAN; PATTABIRAMAN, 2017b).
- ThingsJS: JavaScript-based middleware abstracts many issues from large-scale distributed systems, such as scheduling, monitoring, and self-adaptation, using a rich constraint model, a multidimensional resource prediction approach (GASCON-SAMSON et al., 2018).
- Jade: It is a framework that allows a developer to mix C and JavaScript, and

the result is a hybrid language to develop IoT applications (GHOSH; JIN; MAHESWARAN, 2014).

- ThingsMigrate: It is middleware for the migration of stateful JavaScript applications across IoT devices (GASCON-SAMSON et al., 2018).
- Opel: It is an IoT platform that enables developers to implement several services swiftly and efficiently via JavaScript (LEE et al., 2017).
- Smart Block: It is a visual programming environment. Smart Block enables untechnical users can write their application on this platform quickly (BAK; CHANG; CHOI, 2018).
- iThem: iTherm enables programmers to create complex algorithms that can connect various IoT services and fully utilize the capabilities of a general programming language (WANG et al., 2022).

Although JavaScript has a consolidated status in web and server-side programming, it has been little explored in some contexts, such as IoT, due to hardware requirements. For instance, the device must have sufficient resources to support the JavaScript engine and run algorithms. Therefore, in order to enable the use of the JavaScript language in IoT programming, it is necessary to implement techniques or tools to improve its performance.

2.5 Summary

In this chapter, we have aimed to provide the reader with a comprehensive understanding of the essential basic concepts to facilitate the comprehension of the proposed improvement approaches.

We start by introducing the concept of embedded systems and explaining the differences between interpreted and compiled languages. Furthermore, we present the Internet of Things as a variant of embedded systems. Moving on, we delve into the JavaScript language and explore its ecosystem, including the virtual machine. Next, we provide an in-depth overview of the XS engine by Moddable, highlighting its unique features.

Finally, we presented a systematic mapping review that includes the latest research on the use of JavaScript in the IoT context. The literature review offered a comprehensive overview of the research area involving JavaScript and the Internet of Things. It revealed new opportunities and challenges that could be explored further. Also, it allowed us to analyze the scenario comprehensively and propose contributions that can advance knowledge and collaborate with the scientific community.

3 ANALYZING JAVASCRIPT CODES

In the context of embedded systems, the trade-off between performance and software development strategies, which need to cater to software requirements, is conflicting due to device-constrained properties (GRESSL; STEGER; NEFFE, 2019). It means that certain structures, functions, and paradigms cannot be used, requiring particular analyzes of source code to achieve the balance between development and resource consumption. Therefore, investigating the behavior of a programming language becomes essential to understand all scenarios of performance and resource consumption.

This chapter analyzes the performance of the JavaScript language in IoT applications. We compare applications written in JavaScript and the C language to determine the resource consumption of each language. The analysis provides practical insights into developing applications for embedded contexts using an interpreted language. Furthermore, it offers valuable information regarding possible enhancements and optimizations to reduce resource consumption.

3.1 Hardware and Software Setup

This section provides an overview of the hardware and software used in this study. We describe the equipment that is shared across all experiments. Specific components such as sensors or actuators are defined in the experiments in which they are utilized.

Given the high heterogeneity of IoT devices in terms of resources, we used the RFC 7228 (RFC 7228: TERMINOLOGY FOR CONSTRAINED-NODE NETWORK, 2023) classification as a reference for selecting devices for the experiments. The RFC 7228 document categorizes constrained devices into three classes based on their computational capabilities. We specifically chose microcontrollers that shape into Class 2, comprising devices with limited memory and processing capacities and networking access.

The empirical studies were conducted using ESP8266 (ESP8266 NODEMCU-12E, 2023), ESP32 (ESPRESSIF SYSTEMS, 2023) as the microcontrollers. These are

popular development boards integrated with a Wi-Fi communication chip; table 5 shows their technical specifications. According to the RFC 7228 parameters, these MCUs are considered constrained IoT devices.

Table 5 – Microcontroller specifications.

Specification	ESP8266	ESP32
MCU	Xtensa single-core 32-bit L106	Xtensa dual-core 32-bit LX6
Wi-Fi (802.11 b/g/n)	HT20	HT40
Bluetooth	-	4.2 BLE
Typical Frequency	80 MHz	160 MHz
SRAM	160 kB	521 kB
Flash	16 MB	16 MB
GPIO	17	36
Hardware/Soft. PWM	-/8 Channels	1/16
SPI/I2C/I2S/UART	2/1/2/2	4/2/2/2
ADC	10-bit	12-bit

Regarding code quality, we conducted a static code analysis using the Halstead-Metrics-Tool (SOFTWARE ENGINEERING RESEARCH GROUP AT POLITECNICO DI TORINO, 2023) to collect Halstead Metrics, and the Lizard tool (TERRY YIN, 2023) to collect Cyclomatic Complexity. These tools allowed us to evaluate different languages that use distinct paradigms based on the statements used in the algorithms from the same viewpoint. Table 6 describes the other software used to support the investigations.

Table 6 – Technical software details.

Software (version)	Description
Espruino engine (2.09)	JavaScript engine and firmware packaging
Espruino IDE (0.74.1)	Web IDE to develop the JavaScript application
Arduino IDE (1.8.13)	IDE to develop the C application
ArduinoJson (6.0)	JSON library for embedded C application
Halstead-Metrics-Tool (1.0)	Tool for static code analysis (C/JS)
Lizard (8.9)	Tool for Cyclomatic Complexity Analysis (C/JS)
rc-switch (2.6.4)	Library for management Radio Frequency (RF) commands for C language
Moddable SDK (3.8.0)	JavaScript engine and firmware packaging
Power profile (3.5.5)	Application for use with Nordic Power profile kit to measure power consumption
ESP-IDF (4.4.3)	Espressif IoT Development Framework to compile ESP8266/32 version

Concerning power consumption, we have chosen the Power Profile Kit II (PPK) (NORDIC SEMICONDUCTOR, 2023) to measure energy consumption. The PPK can be operated as an ampere meter or source measure unit (SMU), with a sampling rate of 100,000 kilo samples per second. The Power Profile Kit enables the measurement of low sleep currents, higher active currents, and short current peaks. We used PPK

to supply the MCU at 5V. The input pin (VIN) and negative pin (GND) of MCU are connected to the positive voltage outlet and ground on the PPK. Thus, the SMU allows us to measure the current flow and report the data on the specific software provided by the vendor of PPK.

Moreover, we perform code-synchronized measurements to determine the exact consumption of each experiment. The PPK can convert digital inputs into a logic analyzer. Therefore, before starting the specific execution, the device can send a digital signal to mark the precise entry point where the algorithm will begin. Then, in the end, another signal is fired, delimiting the area of interest for analysis. Figure 15 illustrates the code-synchronized markers.

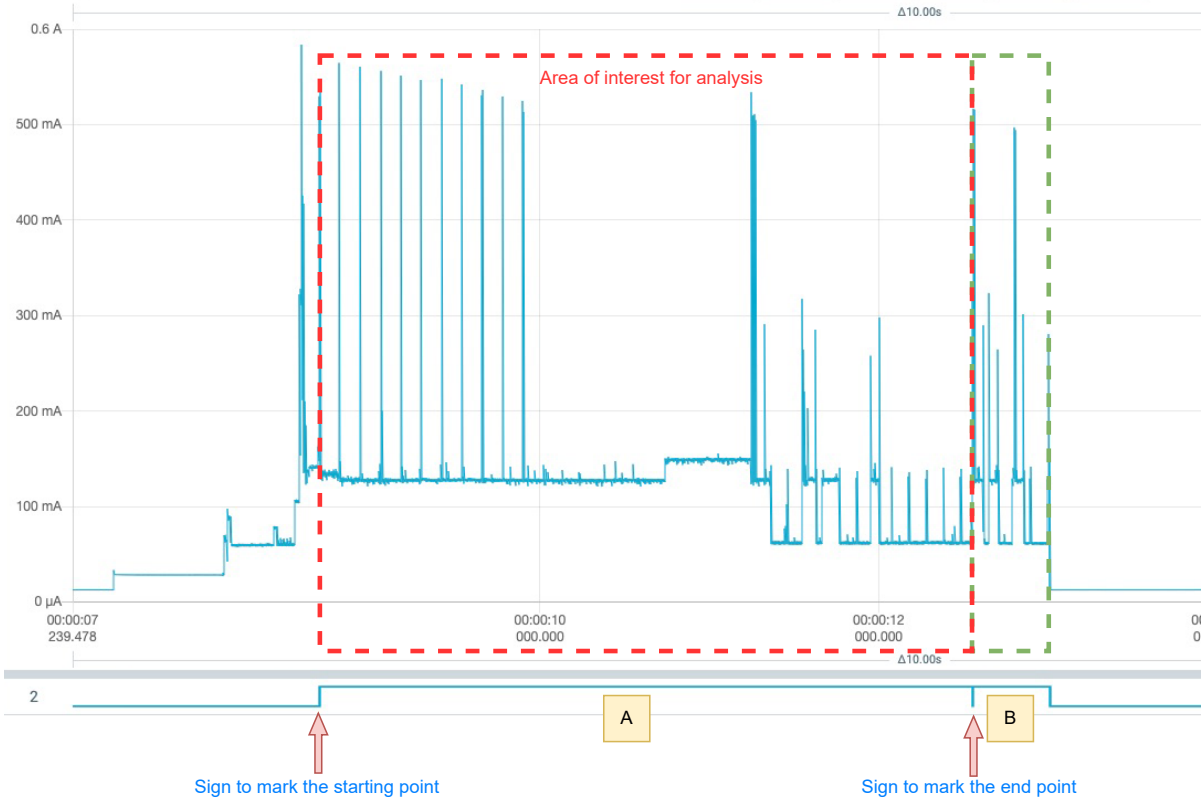


Figure 15 – Code-synchronized measurements example.

Figure 15 shows the threshold of the code being measured for power consumption. In particular, this figure presents two areas defined for analyzing energy. Section (a) represents an execution with a duration of $\approx$ 3.5 seconds, while section (b) represents $\approx$ 0.5 seconds. This example uses logic port number 2 to control the signals, and before starting the specific routine, a high signal is sent, and at the end, a low signal is transmitted to determine the end of consumption. In special, the end of the section (a) and the start of section (b) may appear to be the same. However, it is just a perception; due to the PPK's high precision, the interest area's delimitation is clearly present in the consumption records. As a reference, each measurement result generates a 1.3 GB file containing approximately 50 million records.

3.1.1 Overview of the Experimentation Flow

The experimentation in this study involved several steps. First, we carefully designed the experiments, including selecting the appropriate IoT devices and underlying sensors/actuators, setting up the experimental environment, and defining the variables and metrics to be measured. Then, we implemented the proposed improvement approaches using JavaScript and compared them with the classic approach. Figure 16 provides an overview of the experimentation flow.

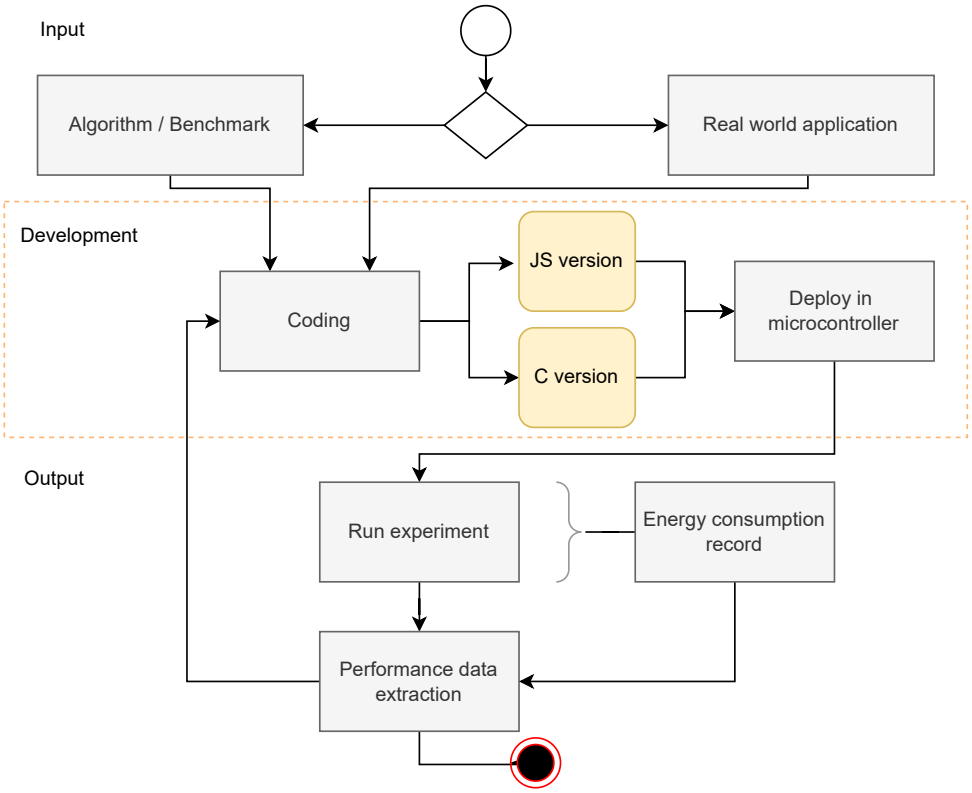


Figure 16 – Overview of the validation flow of experiments.

Figure 16 represents a generic validation flow applied to perform the experiments. Of course, some tests had particular variations according to the goal of the investigation; however, this model can be used to illustrate the overall validation process.

Experiments can be performed using algorithms developed in C and JavaScript, benchmarks, or microbenchmarks. Regardless of the input, they were deployed on the device, and during the execution of the tests, the device was monitored to extract metrics using a data-only cable connected to a personal computer. A power supply was provided by an ampere meter.

We conducted multiple rounds of experiments, carefully controlling and varying the parameters, and collected data on the performance and other relevant metrics. Each experiment was repeated at least 30 times to ensure data consistency and to obtain more accurate statistical information. Thus, the performance data represent the aver-

age of all the executions. Finally, we capture the energy consumption using a dedicated ampere meter.

3.2 Case Study Description

To better understand the relationship between interpreted and compiled languages, we implemented an application to examine the behavior of the JavaScript language, identify any potential limitations or restrictions, and gain an overall understanding of its performance. In order to understand the behavior of a conventional solution, we also developed the same application using the C language. This enabled us to compare the performance and code quality of both applications.

The application consists of a home automation system integrated with the Google Smart Home platform (GOOGLE ACTIONS, 2023). It allows users to control their devices through the Google Home app, Google Assistant, and custom applications. In particular, this automation proposal controls an automatic garage door opener through a 433 MHz frequency. Figure 17 presents an overview of the architectural integration between the in-house software and the Google platform.

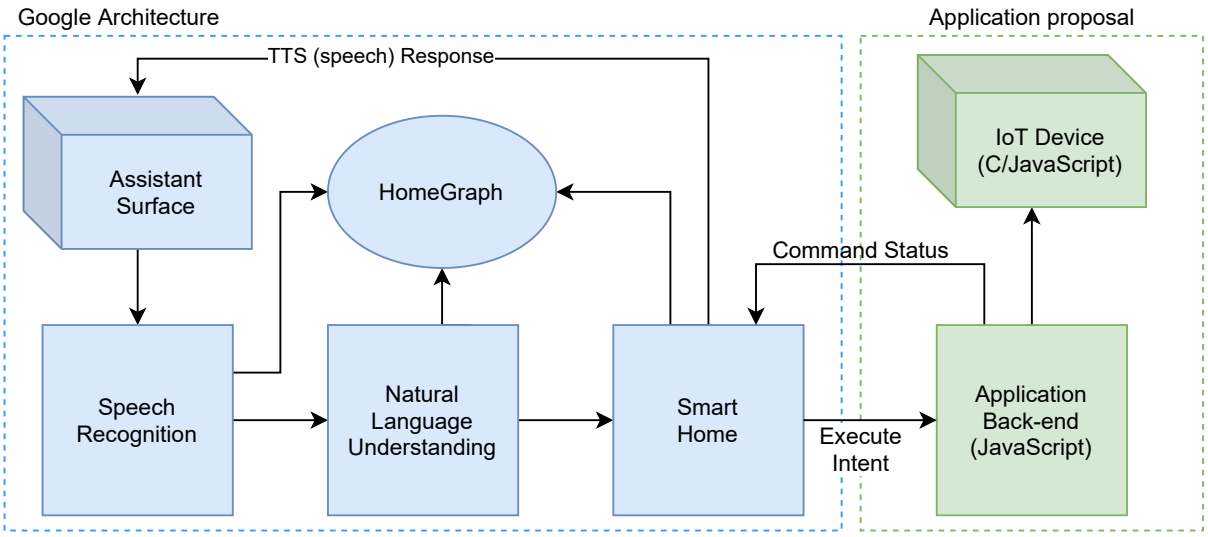


Figure 17 – Garage door opener architecture integration.

Figure 17 shows the integration of the application proposed with the existing (legacy) system. When users send commands to devices with Google Assistant, the server receives an intent that describes the action and the devices to act upon. This intent can be executed on a device via voice commands such as ”Hey Google, open the garage door,” and through custom apps or web interfaces.

Communication between components and devices occurs through specialized messages in a specific format called Google HomeGraph. Each command from the server to the IoT device, and vice versa, is represented as a JavaScript Object Notation (JSON), which describes the request, device, and action to perform. Figure 18 il-

lustrates an example of a message used for system communication.

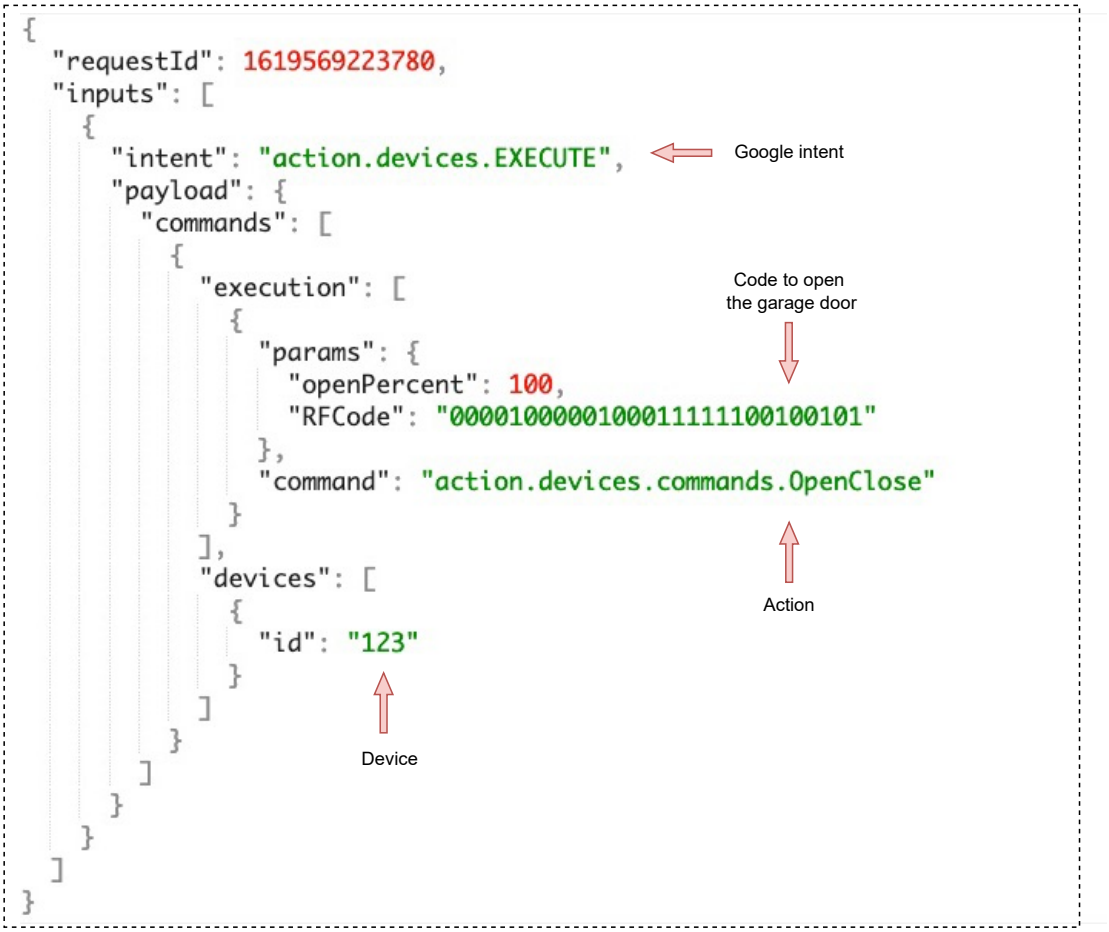


Figure 18 – Google homegraph message.

As illustrated in Figure 18, the Google HomeGraph format describes actions using pre-established constants and can carry out several instructions simultaneously. Furthermore, we included an extra parameter (RFCode) in the original model, which contains the binary code to interact with the garage opener. Finally, we collected the RF code to open and close the door using an RF receiver sensor.

Regarding the Hardware, we choose ESP8266 (ESP8266 NODEMCU-12E, 2023) as microcontroller (MCU) and the FS1000A RF transmitter (COMPONENTS INFO, 2023a) for wireless communication between the IoT device and the embedded garage door circuit. Figure 19 presents an overview of the garage door opener application.

Figure 19 shows that the user can start the interaction by saying a command. It is processed in the Google cloud, which finds an intent and sends it to the application server, and finally, it notifies the IoT device. This process could be initiated without a voice command using the Google Home app or any in-house application. If the user has Google devices on the same network, the command may not need to be processed on the cloud server because these devices can load and run custom applications (written in JS) upon themselves. From that point, the communication is over Wi-Fi, reducing

the latency and increasing reliability (GOOGLE ACTIONS, 2023).

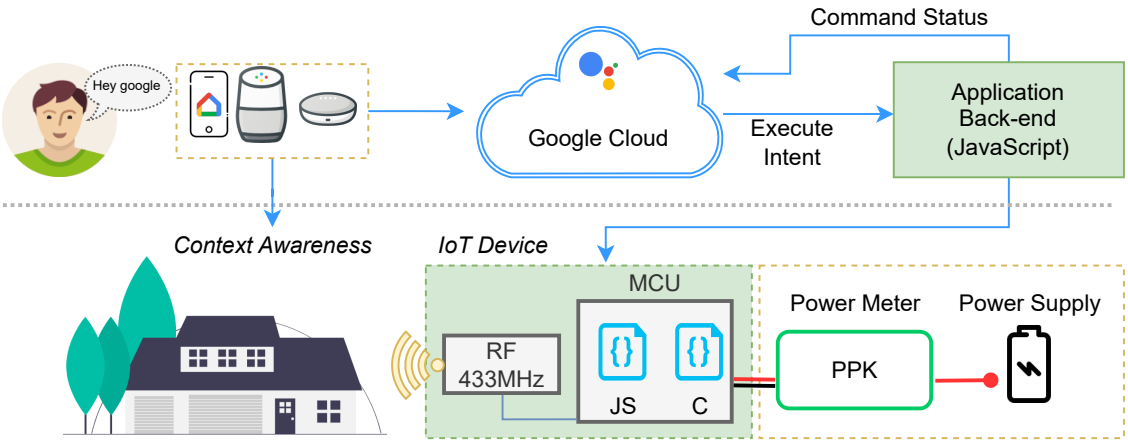


Figure 19 – Garage door opener overview.

For this experiment, the Espruino engine (ESPRUINO, 2023) was selected as JS interpreter. Espruino is an open-source JavaScript engine for microcontrollers with complete coverage of ES5 and partial coverage of ES6 features. Furthermore, Espruino provides firmware with support for the ESP8266 board and a Web IDE for development.

Finally, regarding the performance analyses, we collected runtime and memory usage data using the JavaScript performance API. For the C language, the metrics were extracted from the Espressif IoT Development Framework (ESP-IDF), and the power consumption was recorded using a dedicated ampere meter.

3.3 Results of JavaScript and C programs

Programmers can propose different solutions to the same problem. We developed the algorithms to keep the source code as simple as possible, by defining only the necessary structure and importing the relevant libraries to facilitate development. This section presents the results related to performance and code quality.

The algorithms comprise a web server implementation that receives and processes POST requests. We created two endpoints, RF and STATUS, to send radio-frequency commands and get the device status, respectively. All requests contain a JSON object for data interchange between the application layers, following the Google HomeGraph format.

3.3.1 Code Quality Analysis

Listings 3.1 and 3.2 present chunks of the source code from both algorithms; configuration and device controls have been suppressed, keeping the focus on business logic.

```
// ...
void loop() {
 server.handleClient();
}
bool isOpened() {
 return digitalRead(magnet_switch) == HIGH;
}
void handlerStatus() {
 if ( isValidRequest() ) {
   const size_t bufferSize = 2 * (JSON_ARRAY_SIZE(2) +
 JSON_OBJECT_SIZE(6));
   DynamicJsonDocument requestArgs(bufferSize);
   deserializeJson(requestArgs, server.arg("plain"));

   DynamicJsonDocument doc(128);
   doc["success"] = true;
   doc["requestId"] = requestArgs["requestId"];
   doc["isOpened"] = isOpened();
   sendResponse(doc);
 }
}
void handlerRF() {
 if ( isValidRequest() ) {
   const size_t bufferSize = 2 * (JSON_ARRAY_SIZE(4) +
 JSON_OBJECT_SIZE(11));
   DynamicJsonDocument requestArgs(bufferSize);
   deserializeJson(requestArgs, server.arg("plain"));
   DynamicJsonDocument cmd = requestArgs["inputs"][0]["payload
 "]["commands"][0]["execution"][0];
   const char* action = cmd["command"];
   const int perc = cmd["params"]["openPercent"];
   const long RFCode = cmd["params"]["RFCode"];
   if ( strcmp(action, "...commands.OpenClose") == 0   ) {
    bool executed = false;
    String message;
    if ( perc == 100 ) {
      if ( !isOpened() ) {
        executed = performRF(RFCode);
        message = "Opening the door";
      } else {
        message = "Door is already open";
```

```
39          }
40        } else {
41          if ( isOpened() ) {
42            executed = performRF(RFCode);
43            message = "Closing the door";
44          } else {
45            message = "Door is already close";
46          }
47        }
48        DynamicJsonDocument doc(128);
49        doc["success"] = executed;
50        doc["requestId"] = requestArgs["requestId"];
51        doc["message"] = message;
52        sendResponse(doc);
53      } else {
54        server.send(400, "text/plain", "Invalid command");
55      }
56    }
57  }
```

Listing 3.1 – Garage opener implementation using C language.

Listing 3.1 represents the C implementation in which the endpoints are defined in the “handlerStatus” method (lines 8-20) and “handlerRF” method (lines 21-56). Entry parameters and return are expected in JSON format, and therefore, it requires manual management of memory space in order to allocate the data (lines 10; 23).

Moreover, Google’s operating model presupposes that the garage door control has an open percentage level. In this sense, we consider the value of 100 percent to open the door, and any value different from this to close it. Listing 3.2 shows the JavaScript implementation.

```
1  // ...
2  const LightExpress = require("light-express");
3  const NodeMcu = require("node-mcu");
4  const node = new NodeMcu();
5  const server = new LightExpress();
6  server.post("/rf", (req, res) => {
7  const cmd = req.body.inputs[0].payload.commands[0].execution
    [0];
8  const result = node.performRF(
9    cmd.command,
10   cmd.params.openPercent,
11   cmd.params.RFCode
```

```
12  );
13  if (result) {
14    result.requestId = data.requestId;
15    res.end(JSON.stringify(result));
16  } else {
17    res.writeHead(400);
18    res.end("Invalid command");
19  }
20  });
21  server.post("/status", (req, res) => {
22   const data = req.body;
23   const result = {
24     requestId: data.requestId,
25     isOpened: node.isOpened(),
26   };
27   res.end(JSON.stringify(result));
28  });
29  server.listen(80);
```

Listing 3.2 – Garage opener implementation using JS language.

It should be noted that the business logic is apparently different between the two algorithms. For instance, the logical process from lines 30 until 47 in Listing 3.1 does not appear in Listing 3.2. This is because the chunk of code was isolated in a common class to be reused in other application layers. Listing 3.3 exposes the standardized code.

```
1   class ExecutionHandler {
2    performRF(action, perc, RFCode) {
3     if (action == "...commands.OpenClose") {
4       let message;
5       let executed = false;
6       if (perc == 100) {
7         if (!this.isOpened()) {
8           this.sendRF(RFCode);
9           message = "Opening the door";
10          executed = true;
11        } else {
12          message = "Door is already open";
13        }
14      } else {
15        if (this.isOpened()) {
16          this.sendRF(RFCode);
```

```
17            message = "Closing the door";
18            executed = true;
19          } else {
20            message = "Door is already close";
21          }
22        }
23        return {
24          success: executed,
25          message: message,
26        };
27      }
28    }
29  }
```

Listing 3.3 – JavaScript standard class.

The class ExecutionHandler encapsulates the application logic, and it should be noted that the methods “isOpened” and “sendRF” do not exist in this class. In this case, we follow best practices for development by adopting the Template Method design pattern, aiming at a specific implementation in a separate class. This is essential to promote flexibility and contribute to interoperability among layers since each microcontroller may have distinct hardware specifications. Finally, Listing 3.4 shows the specific board class.

```
1  class NodeMCU extends ExecutionHandler {
2    isOpened() {
3      return digitalRead(D2) == 1; ;
4    }
5    sendRF(code) {
6      require("RcSwitch").connect(6, D4, 3).send(code, 28);
7    }
8  }
```

Listing 3.4 – JavaScript ESP8266 class.

Listing 3.4 represents the custom implementation for the ESP8266 board. For instance, if we needed to implement it on another microcontroller, such as Raspberry Pi, we would only need to create one class to provide the abstract methods, and the rest of the structure could be reused. Table 7 presents the static code analysis of the algorithms; the values inside the table represent a number on a magnitude scale according to each criterion.

As before illustrated, the measures shown in Table 7 point to the overall quality of the produced programs. We conducted the code analysis over the C file and all JavaScript files; the JavaScript measurements are the average ones. In contrast to

Table 7 – Garage opener: Halstead metrics.

Metric	C	JavaScript
Program length	803.0	440
Program vocabulary	140	100
Estimated length	905.56	567.29
Purity ratio	1.13	1.29
Volume	5724.81	2923.30
Difficulty	52.03	45.00
Program effort	297887.75	131548.35
Time to program (h)	4.6	2.03

the performance examination, JavaScript surpasses C in all items, and in some cases, the results can be almost twice as much. Next, we describe each index used in the Halstead metric:

- **Program length:** The size of the program;
- **Program vocabulary:** Number of operators and operands that compose the program;
- **Estimated length:** Metric of size estimated by removing everything from the program except operators and operands;
- **Purity ratio:** This metric assesses the code length based on its actual length. This is an optimization indicator, so the lower the ratio is, the greater the chance that excessive code implements functionality, and a higher ratio (above 1.00) indicates optimized code;
- **Volume:** Measures the size of the implementation of an algorithm;
- **Difficulty:** Difficulty level or error proneness of the program is calculated from the number of unique program operators;
- **Program effort:** Represents the mental activity a programmer performs to transform an algorithm into a program;
- **Time required to program:** Represents the time to develop or understand a program.

Regarding the cyclomatic complexity index, this measure represents how complex each code was considered. Empirical verification shows that the C algorithm had a higher complexity index than the JS implementation. Figure 20 shows the detailed complexity analysis.

From Figure 20, the acronyms mean NLoc (Number of lines of code without comments), Token (Token count of functions), and CNN (Cyclomatic Complexity Number).

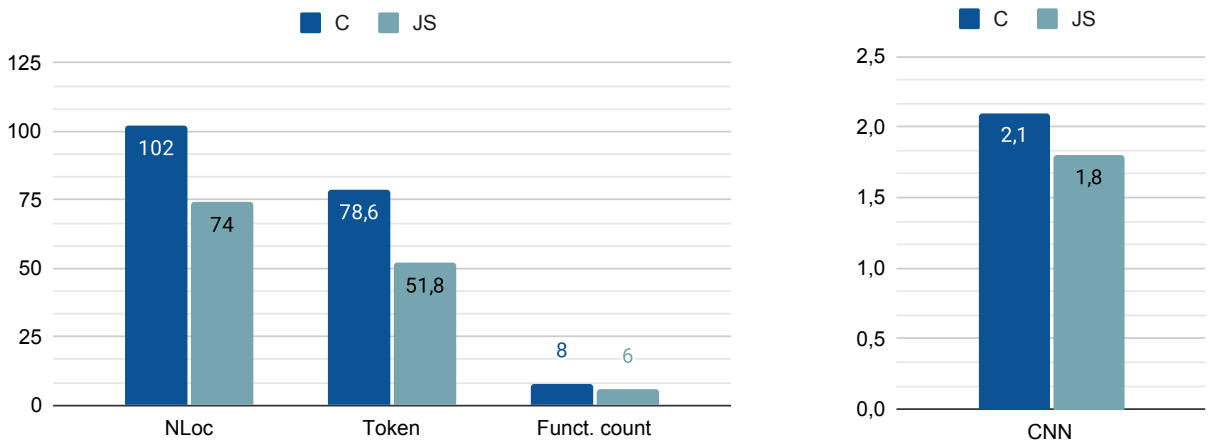


Figure 20 – Garage opener: Cyclomatic complexity analysis.

Although the advantage of JavaScript over the C language is clear, it was conducted from a source developed and proposed by us. Thus, we decided to apply the same tools over an external set of algorithms to compare the results and find some behavior patterns. For that, we selected the Ostrich Benchmark Suite (KHAN, FAIZ AND FOLEY-BOURGON, VINCENT AND KATHROTIA, SUJAY AND LAVOIE, ERICK, 2014). This benchmark provides some facilities for evaluating JavaScript against C because it gives the same implementation of the algorithm in both languages.

The results of the code analysis from the Ostrich benchmark (Appendix B) show that JavaScript exhibits balanced or superior behavior. This analysis is consistent with the results obtained from the code produced in this experiment, which gives us confidence in the achieved outcome.

3.3.2 Resource Consumption Analysis

Assuming that the Alarm System application was built using a web server, the performance results were obtained by executing a predefined set of requests (as previously established, 30 in total). Table 8 lists the details of memory usage.

Table 8 – Garage opener: Memory consumption (bytes).

Language	Memory used (%)	Free memory	Available memory	Total memory
C	5924 (11,09)	47512	53436	96000
JS	19880 (62,12)	12120	32000	96000

The results indicated significant performance variations in memory consumption. The JavaScript solution attained a much higher memory consumption compared to the C algorithm. Specifically, JS consumed more than three times the amount of memory to perform the same algorithm. This difference is attributed to the overhead of the JavaScript engine and its practice of reserving memory space for the entire heap, even

if only a small portion is used.

Although both algorithms were deployed on the same hardware, meaning they have the same amount of memory, the available memory for each algorithm was distinct. Figure 21 shows the difference in the available memory.

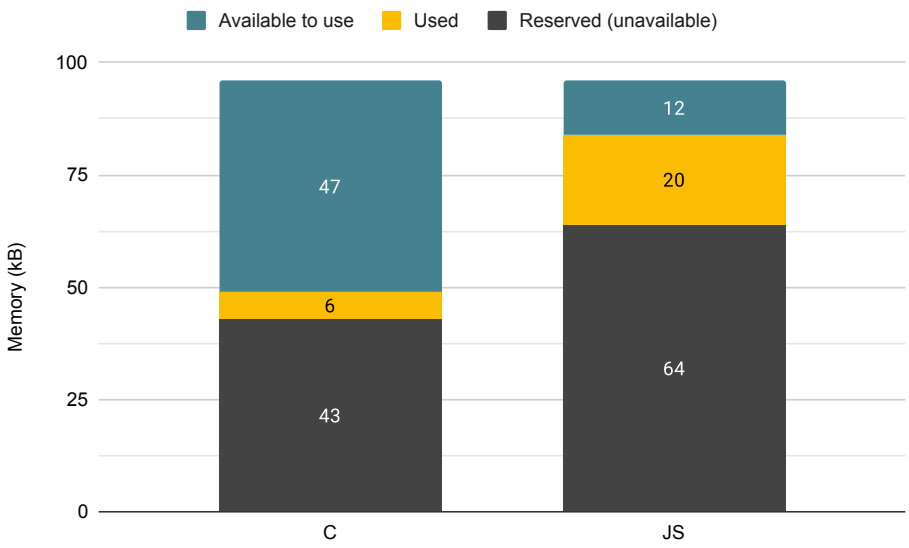


Figure 21 – Comparison of available memory.

The ESP8266 microcontroller has 160 kB of RAM memory, of which 64 kB is allocated to IRAM (for instructions) and 96 kB to DRAM (for data). In this scenario, both solutions start with an equal amount of available memory (96 kB). However, the C solution had $\approx$ 53 kB of available memory, whereas the JavaScript approach had only 32 kB. In both approaches, the firmware (RTOS) and ESP8266 SDK occupy part of the memory space to support Wi-Fi and communication protocols such as TCP/IP.

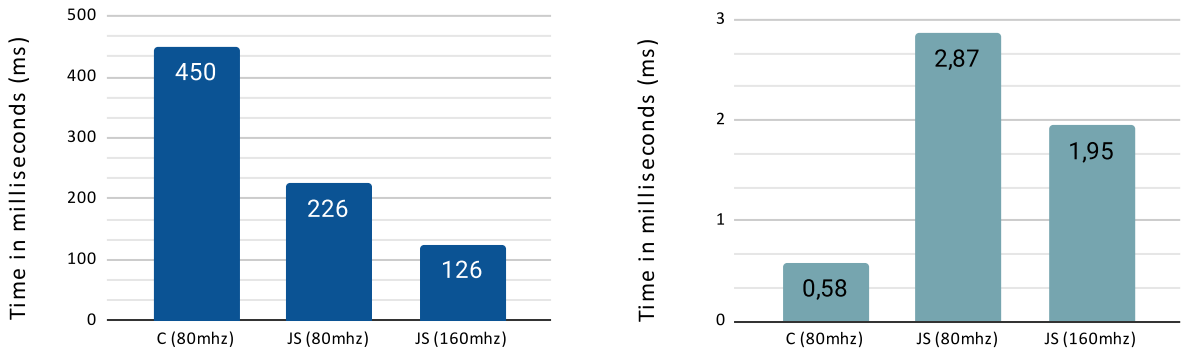
Therefore, we can consider the overhead of the JavaScript solution to be around 21 kB. In other words, using the C solution as a reference, this amount represents 48.84%[1] overhead. This is a significant result because we are working with resource-constrained devices.

Regarding the execution time, we measured how long it took to process each request and clustered the results by algorithms. Figure 22 summarizes the execution time by language.

Figure 22 presents the average execution time for each endpoint. Each graph shows the JavaScript results twice because Espruino sets the ESP8266 clock to 160 MHz by default, whereas the C algorithm operates over 80 MHz. Therefore, we decided to present all the data.

Section “a” in Figure 22 reports the experiment in which the 433MHz RF transmitter is used to open or close the garage door. The C algorithm took longer than the JS

[1] ((21 * 100) = 2100 / 43 = 48.837209302325581

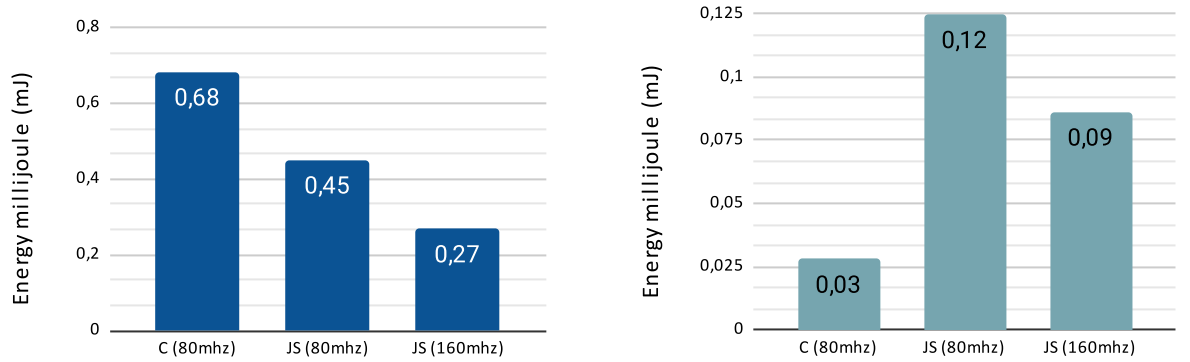


(a) RF request (b) STATUS request

Figure 22 – Garage opener: Execution time.

algorithm. Upon closer analysis, we found that most of the time was spent sending RF commands through the RC-switch library (RC-SWITCH LIBRARY, 2023). Therefore, the extra time is related to a third-party library and not specifically to business rules.

In contrast, section “b” reveals that C is at least four times more efficient than JavaScript at getting the status. After inspecting the source code, we discovered that most of the time was spent on processing requests from the web server. In addition, the I/O operation to read the digital port is as fast as in the C algorithm. Therefore, once again, the slowdown is caused by libraries that are likely to be designed for general purposes and are not optimized for embedded contexts. Finally, Figure 23 presents data regarding energy consumption.



(a) RF request (b) STATUS request

Figure 23 – Garage opener: Energy consumption.

In general, JS consumes more energy than C when performing the algorithms. There are two hypotheses to explain this phenomenon. First, it may be because the JavaScript engine performs optimizations to enhance performance, which leads to direct CPU usage and an additional energy cost. This is a common strategy used by JS engines to improve their performance. Second, extra consumption is likely due to the high memory consumption and slow response time when handling requests from the web server.

Overall, C performed better than JavaScript, as expected. However, the JS application produced a satisfactory outcome that was not too far behind a compiled language and achieved good results regarding design-time metrics. Therefore, the decision to use JS should not be based solely on performance issues but should also consider other aspects of the coding process and post-development, such as maintenance, readability, and code reuse.

3.3.3 Related Work

The majority of research in the literature focuses on code analysis from the perspectives of desktop, server, or web applications. We have selected works that specifically address code analysis in the context of embedded systems.

Joshi; Gurumurthy (2014) investigate optimizations at the source code level for ARM processors in the embedded system context. Their research focuses on loop transformation techniques, and their results indicate a 40

Kienle; Kraft; Nolte (2012) explore static code analysis for a specific industrial embedded system developed using the C language. They present a case study and report insights that can complement more generic code analysis tools.

On the other hand, Oliveira et al. (2008b) investigate the relationship between quality metrics for software products and physical metrics for embedded systems. They use the Java language and establish a strong correlation between quality metrics for traditional software products and performance metrics for embedded systems.

Typically, code analysis approaches for embedded systems focus primarily on device performance, sometimes overlooking software quality metrics. Therefore, it is crucial to evaluate the impact on the source code beyond just performance, considering other perspectives such as the developer or software engineering viewpoint.

In this study, we explore JavaScript as an alternative programming language in the context of the Internet of Things. Additionally, we investigate the advantages and disadvantages of JS compared to the C language, specifically focusing on resource consumption and software quality metrics such as complexity, maintenance, and reuse.

3.4 Summary

The text discusses the analysis of JavaScript codes in the context of embedded systems, particularly in IoT applications. The trade-off between performance and software development strategies is examined due to the constraints of device properties. The chapter aims to find a balance between development and resource consumption by analyzing the behavior of the JavaScript language.

The comparison between JavaScript and C languages is conducted by implementing an application for a home automation system integrated with the Google Smart

Home platform. The architecture and communication flow between the application, Google platform, and IoT devices are explained. The Espruino engine is used as the JavaScript interpreter for the experiments, and runtime, memory usage, and power consumption are measured for performance analysis.

The results of the program comparison are presented, focusing on code quality analysis. Chunks of the source code for both the C and JavaScript implementations are provided. The JavaScript implementation demonstrates better code quality, obtained from Halstead and Cyclomatic complexity metrics. Overall, the analysis provides practical insights into developing IoT applications using JavaScript and highlights possible enhancements and optimizations to reduce resource consumption.

4 JSGUIDE: GUIDELINES TO IMPROVE EMBEDDED SOFTWARE FOR IOT

Embedded development requires developers to have a good working knowledge of hardware and programming languages, as well as an awareness of resource consumption to build efficient applications (KRAELING, 2013). However, finding qualified professionals can be difficult, or in large projects, building efficient solutions can be challenging. An alternative to mitigate this problem is the use of guidelines.

This chapter analyzes the JavaScript language using code smells. We validate well-known code smells established for other contexts, such as desktop, web, or server-side, and suggest new ones based on the latest JavaScript API. Additionally, we introduce JSGuide, a tool for detecting code smells in the IoT context.

4.1 Guidelines

Guidelines can help developers to make algorithms more efficient and lead to the correct use of Application Programming Interfaces (APIs) (KRAELING, 2013).

Developers have been using guidelines to achieve performance, suit code style, satisfy conventions and improve user experience. Thus, guidelines can improve the source code by showing or transforming the chunks of code in order to fix or enhance any aspect (LÓKI.; GÁL., 2018).

Lacerda et al. (2020) argue that finding a source code optimization or code smell entry is a significant challenge for engineering. Also, some items are only found under specific conditions or when combined with or close to other architectural characteristics, making them hard to detect. In addition, code smell detection can be identified through human perception, rule-based strategies, search-based methods, metric-based approaches, and software visualization. The static analysis technique is a way to facilitate the detection of code smells related to the source code.

In this context, this specific study aims to identify and validate code smells for embedded development and automate their detection process. Figure 24 shows an overview of this investigation.

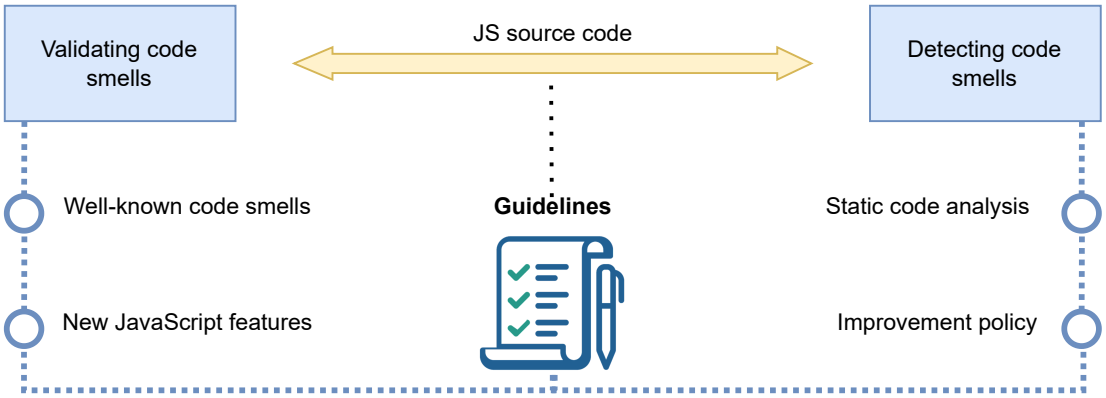


Figure 24 – Goals of the study of guidelines.

This investigation had two goals. First, it aimed to analyze the JavaScript language based on well-known code smells traditionally used in desktop or server-side applications and validate their applicability in the embedded system domain. Second, it sought to create a tool capable of performing static code analysis to detect the bad smells identified in the first step or to detect situations according to specific performance policies.

4.1.1 Selection of code smells

Regarding traditional code smells, we selected the tests inspired by the following studies: Fard; Mesbah (2013), Lóki.; Gál. (2018), and Saboury et al. (2017). Furthermore, we chose tests that could be applied to the embedded system domain, excluding, for example, those suggested for Document Object Model (DOM) because they do not fit into the IoT context. Also, some code smells are simply recommendations to improve code quality, such as avoiding variable re-assignment to enhance readability; thus, they were also discarded. Therefore, we composed a set of tests based on situations that allow us to identify code labeled as bad smell and suggest corrections/improvements. In the following, we present the selected code smells.

- **Arrow function:** To use arrow functions to reduce the syntax applied to scenarios when a non-method function is needed;
- **Binary literals:** Choose to use binary literals avoiding conversion of types;
- **Inline function:** This is a legacy optimization that helps to save function calls instructions;
- **Iterator:** Iterators are recommended because they can provide an optimization way to iterate over all items, saving memory;
- **Long parameters:** Reducing the number of method parameters can increase readability and backward compatibility;

- **Loop iterator:** It helps to reduce execution time by reducing the number of iterations and loop test instructions;
- **Map:** It gives a more suitable structure to manipulate objects that hold key-value pairs;
- **Template string:** It gives JavaScript the ability to create new strings based on a previously defined template;

Beyond validating well-known code smells, we also introduce new ones based on the latest JavaScript specifications. The ECMAScript specification is released annually, typically announced in June. We considered features published from ES9 (2018) until ES12 (2021) for this experiment.

The selection of test cases was based on the specification that presents new ways to develop existing routines or introduces new syntax for well-known structures. For instance, to execute a search for a match in a specified string, we would use the “exec” method combined with a “while” statement to iterate over the results. ES11 introduced a new method called “matchAll” that provides an alternative to searching in strings. Figure 25 demonstrates the difference between JS APIs.

Old API

```
const msg = "sample text";
const regexp = new RegExp(/[^\s]+/g);
let value;
while ((value = regexp.exec(msg)) !== null) {
  console.log(value[0]);
}
```

New API

```
const msg = "sample text";
for (const value of msg.matchAll(/[^\s]+/g)) {
  console.log(value[0]);
}
```

Figure 25 – Comparison between old and new JavaScript API.

In other words, ES11 creates a new way to develop the same existing function through a novel API. In particular, the “matchAll” method returns an iterator instead of an array with all the matches. The iterator is, theoretically, more efficient because it does not load the entire collection into memory to iterate over it. Table 9 summarizes the JavaScript new features analyzed.

To perform the validation of the code smells and produce guidelines, we established a test process. First, we determined the set of code smells and the newest JavaScript

Table 9 – Evaluated JavaScript Features.

Feature	Version	Description
Rest	ES9	Allows a function to accept an indefinite number of arguments as an array.
Spread	ES9	Allows an iterable, such as an array or string expression, to be expanded in places where zero or more arguments.
fromEntries	ES10	The method transforms a list of key-value pairs into an object.
flat	ES10	The method creates a new array with all subarray elements concatenated recursively to the specified depth.
trimStart / trimEnd	ES10	The method removes leading and trailing whitespace from the string.
matchAll	ES11	The method returns an iterator of all results matching a string.
Nullish coalescing	ES11	It is a logical operator that returns its right-hand operand when its left-hand operand is null or undefined and otherwise returns its left-hand operand.
Optional Chaining	ES11	It enables to read the value of a property located deep within a chain of connected objects without checking that each reference in the chain is valid.
replaceAll	ES12	The method returns a new string with all matches of a pattern replaced by a replacement.
Logical assignment	ES12	A new way to represent the logical AND assignment.

features for evaluation, and then we made microbenchmarks. Figure 26 shows the overview of the language analysis process.

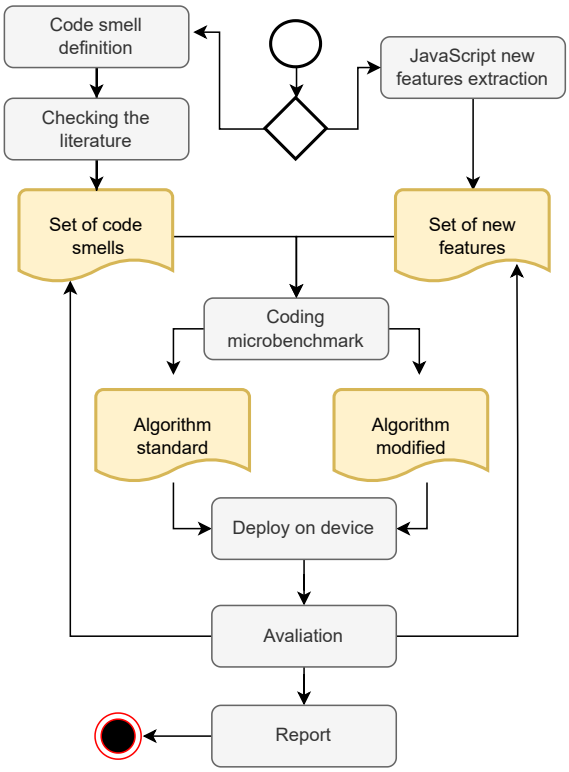


Figure 26 – Overview of the JavaScript language analysis process.

Figure 26 depicts the language analysis process flow. We extract the feature to validate and put it inside a specific class that represents a microbenchmark. Its execution follows the previously established execution flow (30 times) to guarantee the statistical

data. Finally, we extracted the energy data, summarized the findings, and generated the report.

Each microbenchmark consists of a class with *standard* and *modified* methods that expose the original and new or changed code, respectively. Microbenchmarks always have the same input (recreated for each execution) and must produce the same output. These classes are submitted to an in-house measurement framework created exclusively to measure execution time and memory consumption through Moddable's instrumentation API. Listing 4.1 presents an example of a microbenchmark, and Appendix A exposes all microbenchmarks used.

```
1 export class Iterator {
2   static standard() {
3     const arr = [1, 22, 13, 1, -2, 3, 6, 0.78];
4     let sum = 0;
5     for (var i = 0; i < arr.length; i++) {
6       sum += arr[i];
7     }
8     return sum;
9   }
10
11  static modified() {
12    const arr = [1, 22, 13, 1, -2, 3, 6, 0.78];
13    let sum = 0;
14    for (let value of arr) {
15      sum += value;
16    }
17    return sum;
18  }
19 }
```

Listing 4.1 – Iterator smell.

Listing 4.1 shows a microbenchmark to validate which technique is more efficient - a typical loop using a for statement or using a for-each loop. Although the code is encapsulated inside a class, the methods are static, ensuring they are not invoked through an object to avoid overhead while validation occurs.

4.1.2 Guidelines Results

We analyzed the JavaScript language using microbenchmarks to validate code smells and explore new JS features. For that, we used ESP32 as MCU and Moddable SDK to support JavaScript on the device. As a result, we examined eight code smells and ten new features.

The general objective of this analysis is not to establish that the new version is superior to the old one but to evaluate which option is the most optimal in terms of resource consumption. Therefore, we discuss the most significant differences between these approaches. Table 10 summarises the results.

Table 10 – JavaScript language analysis results.

	Mem. (bytes)			Exec. time (ns)			Energy (mJ)		
	STD	MDF	%	STD	MDF	%	STD	MDF	%
Code smells									
Arrow	304	320	5,00	1886,27	2005,03	5,92	0,81	0,86	6,10
binary	0	0	0,00	627,73	399,53	-57,12	0,27	0,17	-57,45
inline function	96	112	14,29	452,67	497,07	8,93	0,20	0,21	8,77
Iterator	168	296	43,24	1951,77	2381,10	18,03	0,84	1,01	17,04
Long parameter	112	112	0,00	497,10	490,00	-1,45	0,21	0,21	-2,30
loop	0	0	0,00	999,80	1033,40	3,25	0,43	0,44	2,73
map	81	260	68,82	1277,83	991,53	-28,87	0,55	0,42	-29,19
template string	200	216	7,41	674,43	684,23	1,43	0,29	0,29	1,26
New features									
Rest	352	208	-69,23	1811,63	1627,80	-11,29	0,78	0,70	-12,10
Spread	128	160	20,00	1153,87	1115,30	-3,46	0,49	0,48	-1,90
From entries	264	392	32,65	1694,73	2405,13	29,54	0,72	1,03	30,19
Flat	1424	560	-154,29	5658,07	2409,33	-134,84	2,43	1,03	-136,10
Trim start	540	52	-938,46	2887,70	676,57	-326,82	1,24	0,29	-326,62
Match all	1760	1964	10,39	6112,93	8014,73	23,73	2,63	3,45	23,76
Nullish coalesing	0	0	0,00	369,40	352,43	-4,81	0,16	0,15	-5,22
Optional chaining	112	112	0,00	749,47	721,93	-3,81	0,32	0,30	-7,70
Replace all	680	296	-129,73	3069,10	1000,60	-206,73	1,31	0,43	-206,62
Logical assignment	0	0	0,00	402,57	399,80	-0,69	0,17	0,17	-1,19

In Table 10, STD and MDF refer to standard and modified codes, respectively. The percentage value is calculated based on the original code. Therefore, a positive value indicates that the original code is better than the modified one, while a negative value indicates the opposite.

First of all, it should be noted that some benchmarks do not have memory data (showing zero). This is not a problem; rather, it is related to the Moddable's approach and its principles for optimizing applications on microcontrollers. In summary, variables can be stored in ROM instead of RAM to improve performance and reduce memory usage. The instrumentation API only measures data in the heap memory; thus, some entries present zero in their results because they were stored in ROM. Moreover, JavaScript's new features yielded good results compared to the results of code smells. Many of these features are more energy-efficient than the previous API. Figures 27 and 28 illustrates the execution between the generations.

One test that performed poorly was the *From entries* code. This function can be seen as a utility function that transforms a list of key-value pairs into an object. The difference in memory consumption was significant at 32.65%, and the execution time was 29.54% longer. Therefore, it is recommended, even though it may be difficult to

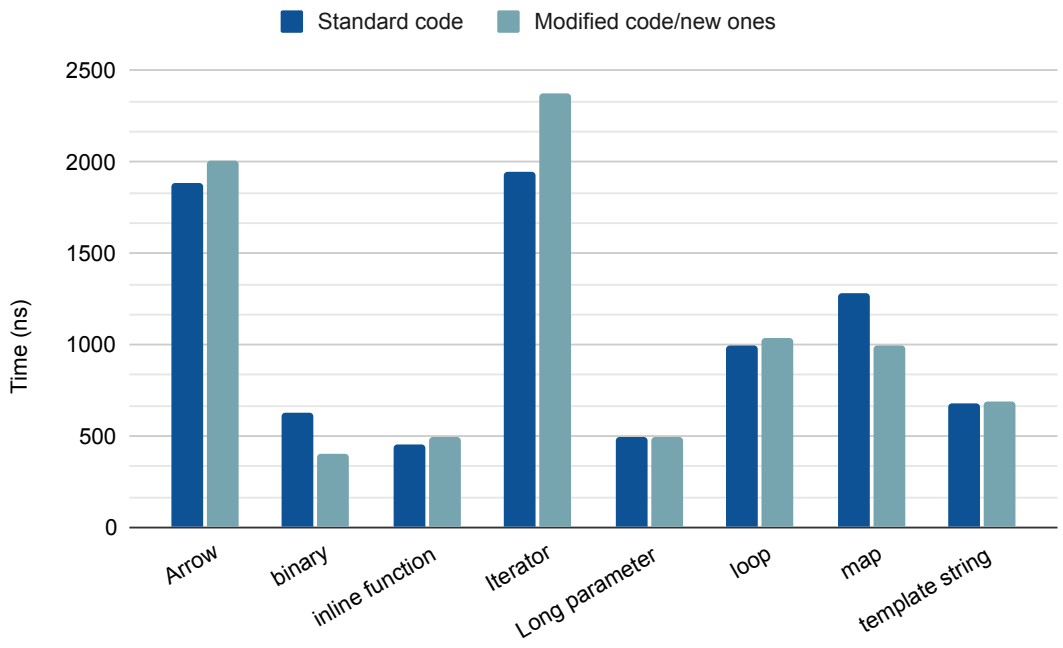


Figure 27 – Code smells: execution time.

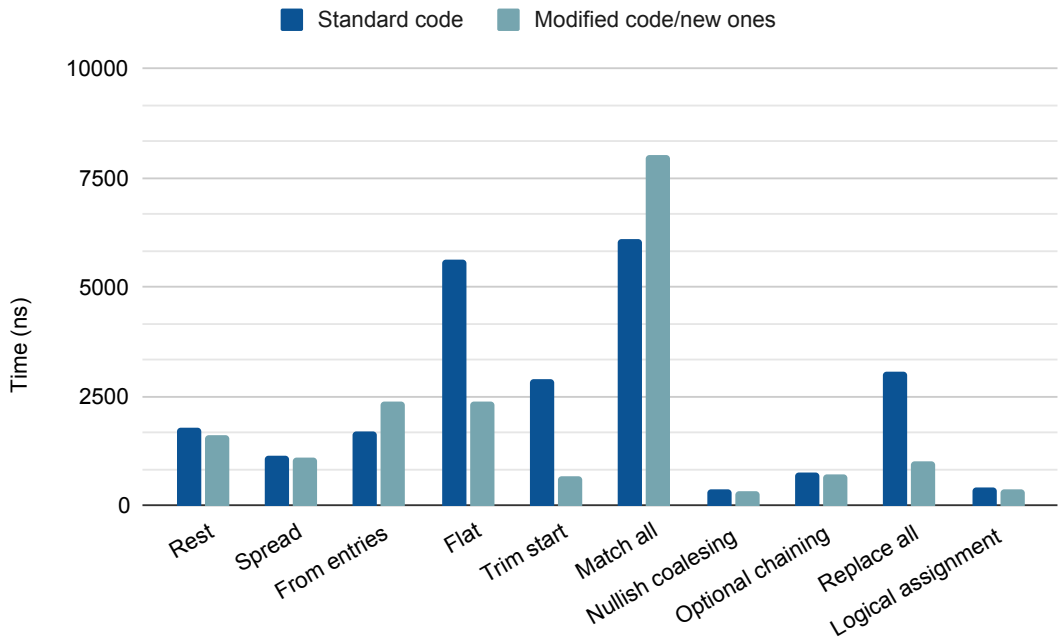


Figure 28 – New features: execution time.

read, to use the classical annotation with key and value.

The fastest tests were *trim* and *flat*. While it represents a simple action, the *trim* test removes empty white spaces at the beginning of a string using a conventional regex-based approach, which significantly impacts performance. On the other hand, the *flat* test uses a specialized implementation to flatten array items to the same level instead of using recursion as in the “reduce” approach.

Figure 28 still indicates that the *match all* method performs slower. Theoretically, this method should be more efficient, but the measures show the opposite trend. This occurs because the structure of the iterator is optimized for scaling larger outputs. For example, instead of returning an array to interact with, the method returns an iterator to handle a few items. Thus, in this experiment, it resulted in slower performance. Figures 29 and 30 illustrates the consumption of memory.

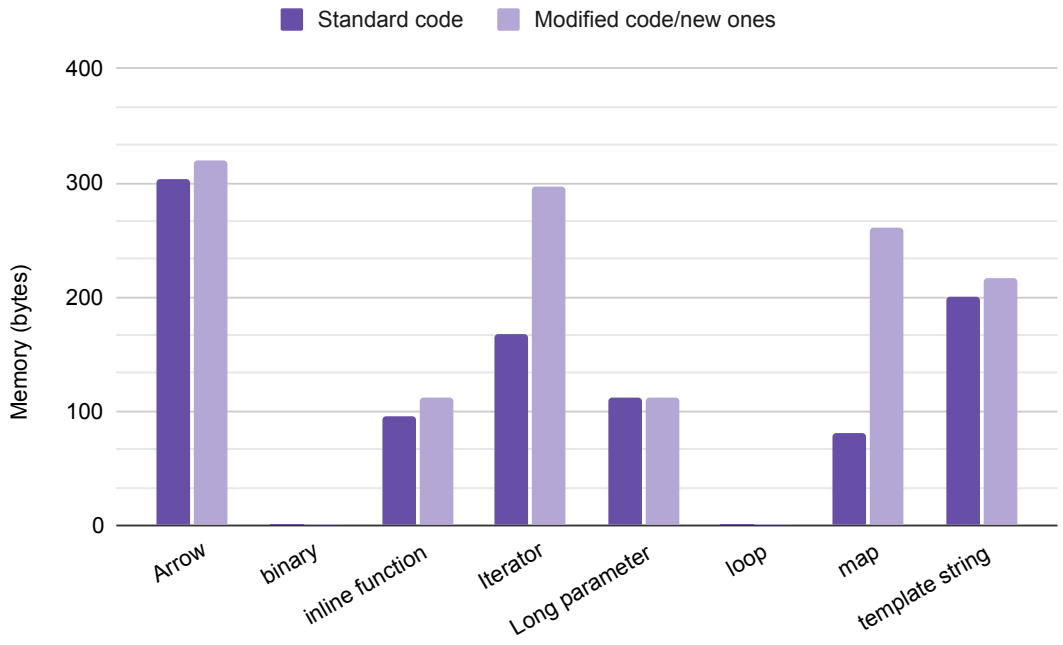


Figure 29 – Code smell: memory consumption.

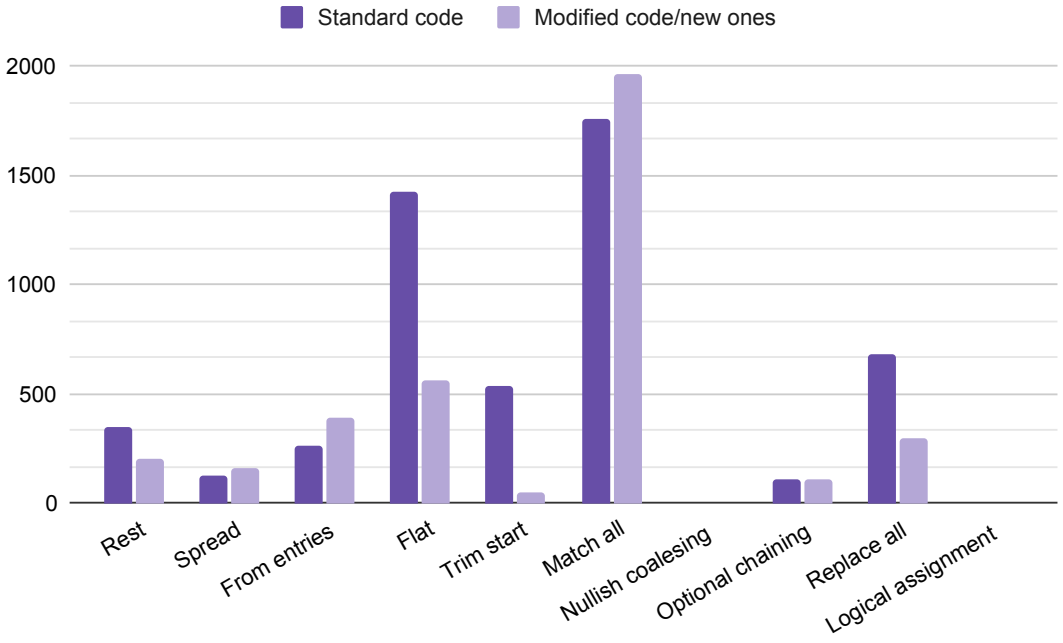


Figure 30 – New features: memory consumption.

Regarding memory, the *iterator* and *map* tests showed the worst consumption. Unlike “for” loops or the “map“ function, the “forEach” method does not provide many optimization opportunities for JavaScript engines to optimize the loop (FLANAGAN, 2020b). This can result in suboptimal performance in some cases. On the other hand, the *map* test presented unexpected results because “map” is often considered to be a concise and memory-efficient mechanism to manipulate array elements.

The *iterator* test measures different ways to iterate over an array. The results indicate that the traditional approach, where the developer has complete control over the execution, is better than the new one. Therefore, the alternative syntax (foreach) should be avoided.

The *template string* test performed worse than the traditional string manipulation. On the one hand, considering the performance aspect, it should not be recommended; although there is a slight performance difference, strings are often used. On the other hand, template strings make the code more explicit, improving readability and maintainability. Therefore, we need to be aware of their costs to decide whether to use them. Figures 31 and 32 shows the consumption of energy..

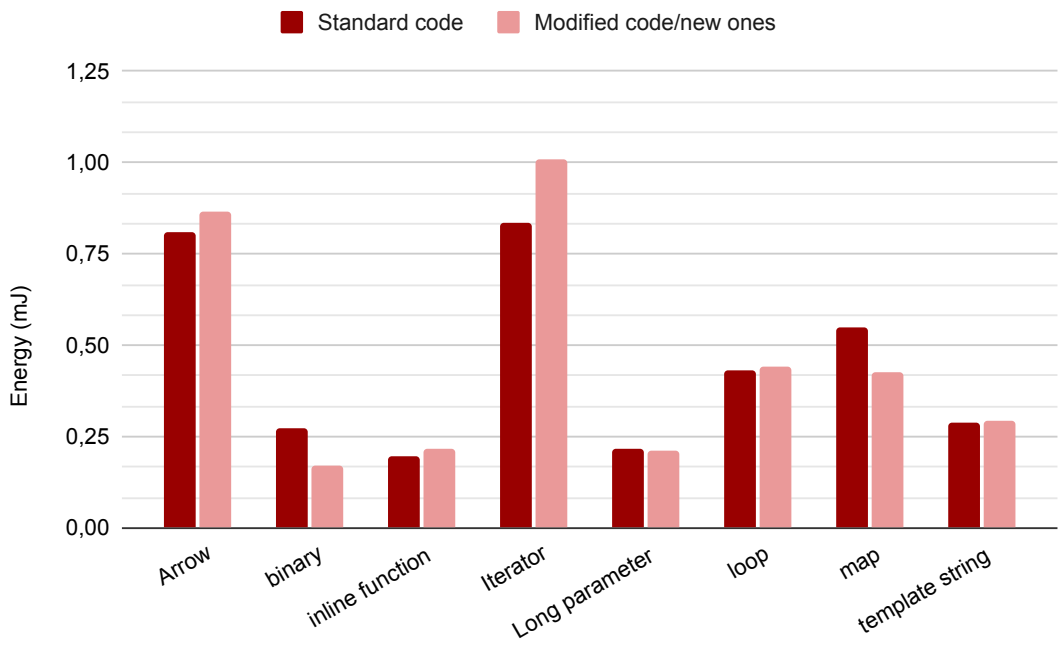


Figure 31 – Code smell: energy consumption.

In general, the results indicated that adopting code smells can contribute to reducing energy consumption. However, codes that utilize regular expressions are slower and tend to consume more energy; therefore, their adoption in resource-constrained environments needs to be carefully evaluated.

Based on the results, if there is even a slight improvement in the algorithm, we would suggest adopting it, as even a small progress can make a significant difference

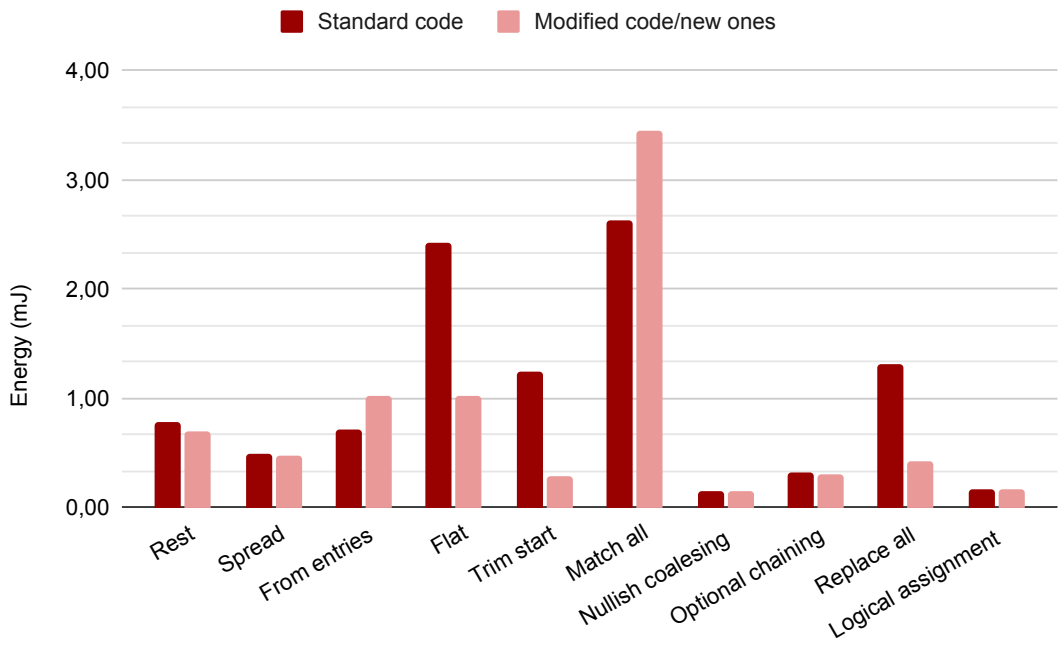


Figure 32 – New features: energy consumption.

on constrained devices. However, it should be noted that these recommendations are dependent on the technology used, such as virtual machines.

Next, we discuss the second part of the guideline study, which is concerned in identifies code smells.

4.2 JSGuide: A tool to detect code smells

In addition to the knowledge gained from identifying code smells, there is still the arduous task of finding code patterns within the source code of applications and depending on the size and scope of the system, this activity can be tiring and prone to errors (FARD; MESBAH, 2013). Therefore, it is essential to adopt a tool to automate this task.

JSGuide is a static code analyzer tool that identifies code smells based on the embedded system context. The framework also uses a heuristic-based approach to detect smells and produces a report suggesting how to improve the embedded software. Furthermore, JSGuide was developed using the Java language and Mozilla Rhino (MOZILLA FOUNDATION, 2022) as a JavaScript parser. Figure 33 provides an overview of the framework architecture.

Figure 33 illustrates the primary pipeline for detecting and reporting code smells. First, the code is submitted to the parser. If the algorithm is parsed, it generates an Abstract Syntax Tree (AST). Next, the Task agent analyzes the AST and produce a report with the recommendations.

The Task agent effectively carries out code analysis. A Task agent is a generic ab-

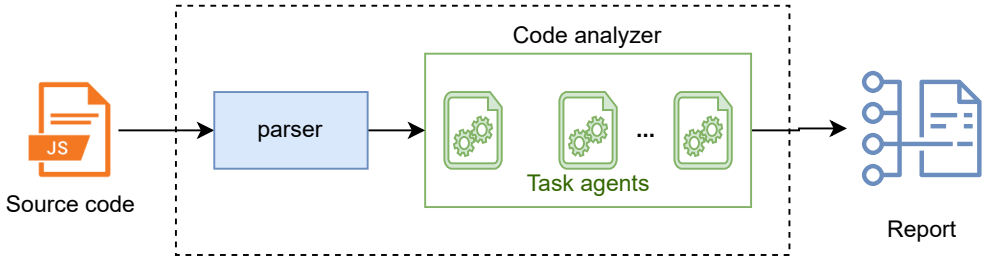


Figure 33 – JSGuide - Architecture overview.

straction that describes the patterns for finding smells. Each agent can be implemented to consider distinct goals. For instance, it is possible to create an agent to improve performance, energy savings, security, or other dimensions according to an optimization strategy. This approach allows flexibility in the evolution or creation of personal profiles to customize applications. Figure 34 shows the flow used to detect smells.

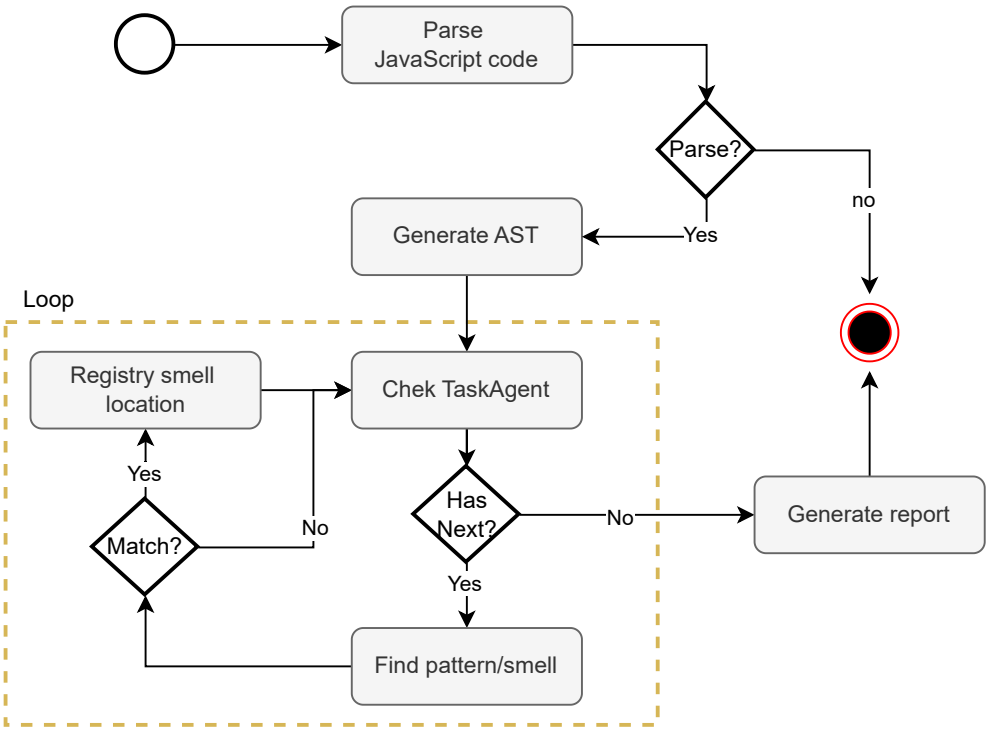


Figure 34 – Code smell detection.

Figure 34 demonstrates the execution flow of the JSGuide algorithm. In summary, the AST elements are analyzed to find chunks of code that match the task agent's criteria. Next, the program iterates over all agents. If any code is consistent with the smell definition, its location (file, line, column) is registered, and a report is generated.

To abstract the detection of code smells or patterns in the code, we created an interface to standardize the actions for finding the code and another one to model the results, considering all the information needed to locate the code snippets in the report. Figure 35 represents a class diagram of the JSGuide.

The interfaces *ILocation* and *ITaskAgent* abstract the common behavior used to

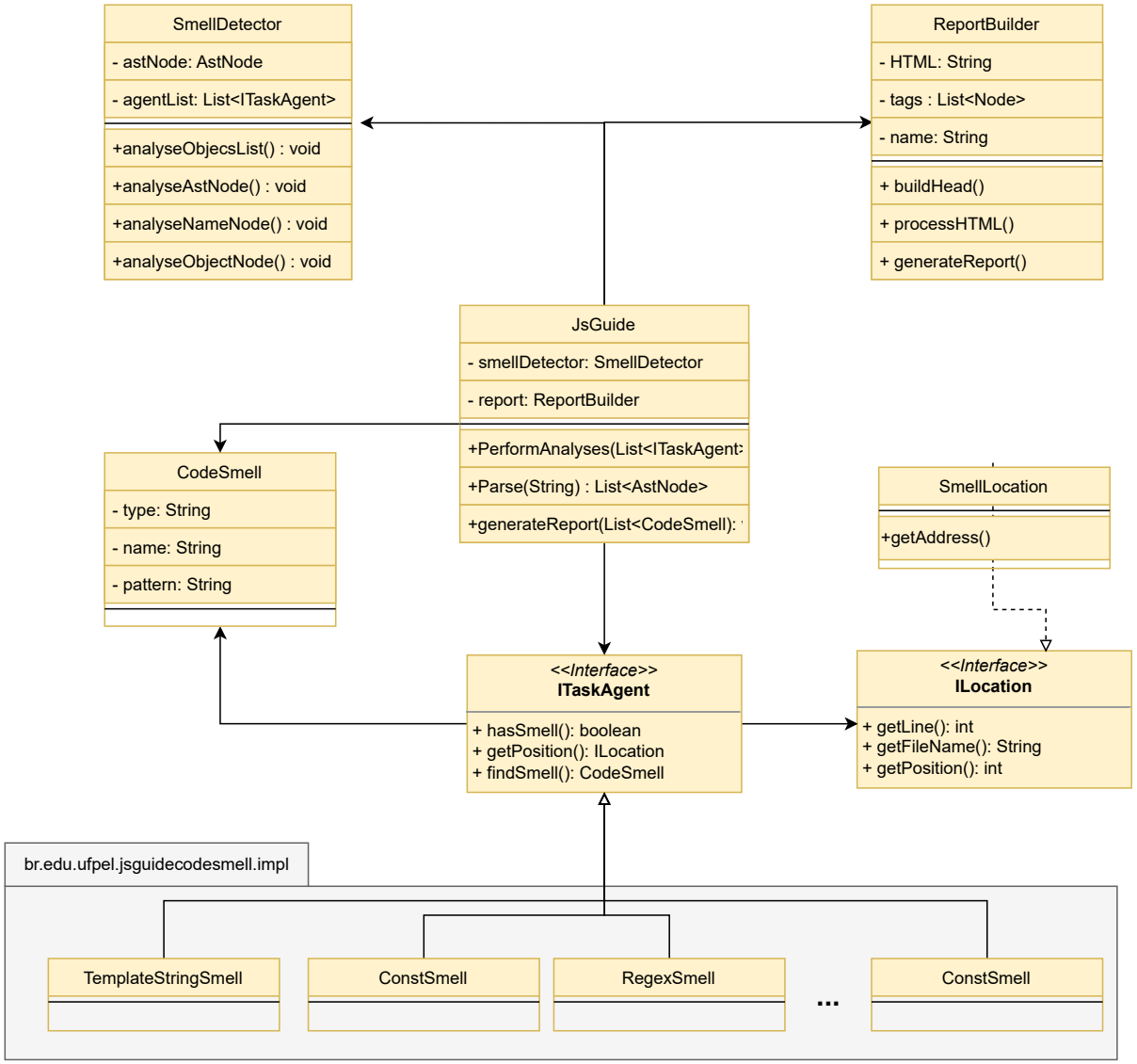


Figure 35 – JSGuide class diagram.

detect and identify code smells. The class *JGuide* represents the executable (main) from which the code is parsed, and the report is built. Moreover, the class *SmellDetector* is responsible for analyzing the AST nodes and validating the TaskAgents. Finally, the classes inside the package "br.edu.ufpel.jsguidecodesmell.impl" represent specific implementations to find particular code smells.

We designed the tool based on interface abstraction instead of concrete classes, which enhances code modularity, extensibility, and maintainability. This implementation promotes loose coupling between components and allows for easier substitution of implementations. Moreover, it enables developers to switch out implementations without affecting the rest of the codebase, facilitating flexibility and adaptability.

4.2.1 JSGuide Results

The basic operating principle of JSGuide is to analyze the source code of the program (static source analysis), looking for patterns compatible with the metrics established in each task agent. When a pattern is identified, the location of that item is recorded and later highlighted in the report.

Therefore, to validate JSGuide, we analyzed benchmark code extracted from The Computer Language Benchmarks Game (CLBG) (THE COMPUTER LANGUAGE BENCHMARKS GAME , 2023) and scripts used in our in-house embedded projects. Listing 4.2 shows the code for the temperature reader.

```
1  import WiFi from "wifi";
2  import Analog from "pins/analog";
3  import { Request } from "http";
4  new WiFi(
5    {
6      ssid: "xxx",
7      password: "xxx",
8    },
9    function (status) {
10     if (status == WiFi.gotIP) {
11       // network is ready
12       let value = readTemp();
13       sendData( value)
14     }
15   }
16 );
17 function sendData(temp) {
18   let request = new Request({
19     host: "api.thingspeak.com",
20     path: `/update?api_key=<HIDE>=${temp}`,
21     response: String,
22   });
23   request.callback = function (message, value, etc) {
24     if (Request.responseComplete == message) {
25       // go to sleep
26       sleep();
27     }
28   };
29 }
30 function readTemp() {
31   let readedValue = Analog.read(0);
```

```
32   return ((readedValue / 1024.0) * 3300) / 10;
33 }
```

Listing 4.2 – JavaScript temperature reader.

Listing4.2 shows the in-house code that performs periodic temperature readings and sends them to the network. This example is interesting because it reads the sensor data, sends it over the network, and then goes into the sleep mode. Putting the MCU in idle mode is a typical strategy to save energy because it allows components such as the modem, sensor, and other elements to be turned off, thereby conserving energy. Figure 36 illustrates the report generated for this code.

Figure 36 – JSGuide: report example for Listing 4.2.

Figure 36 illustrates the produced report, which is built using HTML and CSS and can be accessed via internet browser. The identified code smells are presented (lines 12, 20, and 31) and described to help the developer understand the issue and promote refactoring. Listing 4.3 shows a benchmark code from CLBG.

```
1 const fs = require('fs');
2 function mainThread() {
3     const regExps = [
4         /agggtaaa|tttaccct/ig,
5         /[cgt]gggtaaa|tttaccc[acg]/ig,
6         /a[act]ggtaaa|tttacc[agt]t/ig,
7     ];
8     let data = fs.readFileSync('/dev/stdin', 'ascii');
```

```
 9     const initialLen = data.length;
10     data = data.replace(/^>.*\n|\n/mg, '');
11     const cleanedLen = data.length;
12     for (let j = 0; j < regExps.length; j++) {
13         const re = regExps[j];
14         const m = data.match(re);
15         console.log(re.source, m ? m.length : 0);
16     }
17     const endLen = data
18         .replace(/tHa[Nt]/g, '<4>')
19         .replace(/aND|caN|Ha[DS]|WaS/g, '<3>')
20         .length;
21     console.log(`\n${initialLen}\n${cleanedLen}\n${endLen}`);
22 }
23 mainThread();
```

Listing 4.3 – CLBG JavaScript exemple.

The benchmark represented in Listing 4.3 is not embedded code, but rather a server-side script. However, the framework can be applied without restrictions, but the results are considered based on its execution over the embedded domain. Figure 37 presents the detailed report.

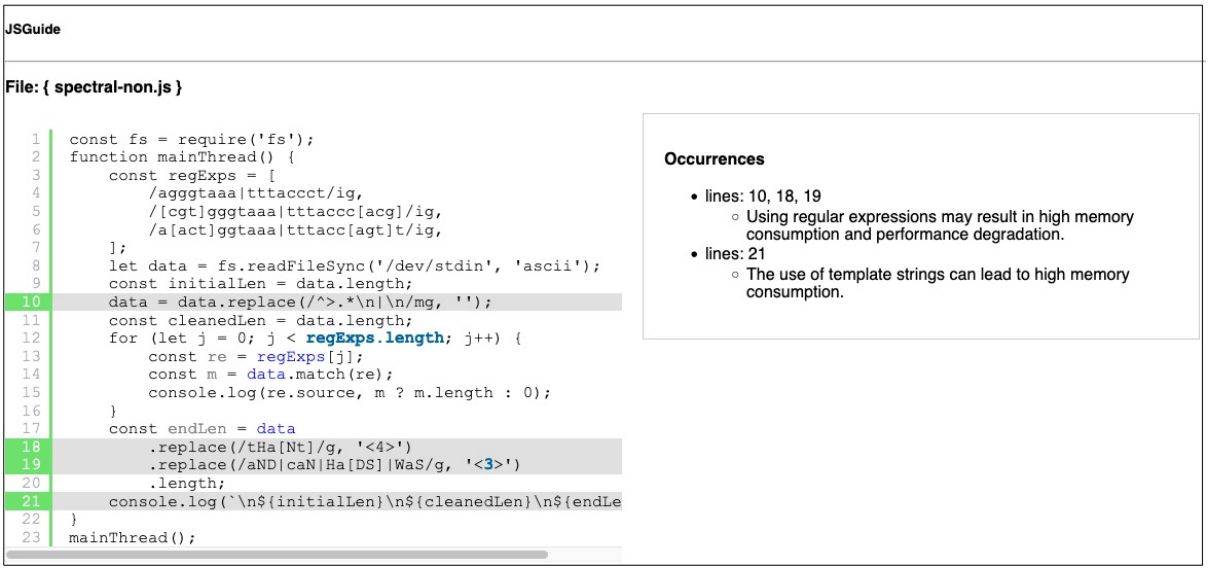


Figure 37 – JSGuide: CLBG report example for Listing 4.3.

The second report highlights restrictions related to the use of regular expressions. Overall, the framework worked as expected, but it required the developer's judgement regarding adopting suggestions and performing code refactoring.

Adopting these suggestions can lead to better code quality, affecting design-time metrics such as readability, maintainability, and code reuse, as well as design-runtime

factors such as performance and energy consumption. Furthermore, the level of improvement may vary depending on the number of code smell occurrences. Therefore, even if refactoring goes unnoticed in terms of general application performance, a minor enhancement can still significantly impact resource-constrained devices.

4.3 Related work

Due to the origins of JavaScript, the majority of works found in the literature primarily focus on guidelines for the web context. Therefore, we have selected works that discuss guidelines associated with server-side or IoT applications.

Lóki.; Gál. (2018) investigated JavaScript performance issues in relation to ECMAScript 6 elements, comparing them with ECMAScript 5.1 variants in order to provide guidelines and optimizations. They analyzed multiple JS engines, ranging from server to embeddable engines, and examined some legacy guidelines. The authors emphasize the importance of using guidelines, although they note that the results for embeddable JavaScript engines are inconclusive due to the engines not fully supporting the spectrum of ECMAScript 6. They recommend reviewing the guideline results as they depend on the evolution of JavaScript engines.

Based on open-source projects, Liu (2019) propose a static analysis framework called JSOptimizer. This framework performs analyses on JavaScript source code, searching for nine performance issues and potential bugs. Additionally, the tool allows for automatic code changes. The experiments were conducted on a desktop computer, although the specific JavaScript engine used is not specified. The author reports an improvement in speed of approximately 300

Fard; Mesbah (2013) presented a tool for detecting code smells in JavaScript. They employed metric-based smell detection through static and dynamic analyses, and proposed 13 JavaScript code smells. The empirical evaluation revealed that lazy objects, long functions, closures, and coupling between JavaScript, HTML, CSS, and global variables are the most prevalent code smells.

Given the context, JavaScript guidelines and code smells are relatively unexplored in the IoT domain. Therefore, this work distinguishes itself by providing an extensible framework for analyzing JavaScript code, specifically targeting code smells and optimization/bug entries within the embedded context. Additionally, we incorporate guidelines based on the latest ECMAScript API.

4.4 Summary

This chapter discusses the use of guidelines to improve embedded software development. In addition, the chapter also explains the methodology used to validate the

code smells through microbenchmarks and discusses the results, including execution time, memory consumption, and energy consumption.

Furthermore, the text introduces JSGuide as a static code analysis tool for detecting code smells in embedded software. It describes the tool's architecture, which is based on Java and the Mozilla Rhino framework. The tool uses a heuristic-based approach to identify code smells and provides a report with suggestions for improving the software.

5 JSEVASYNC: A FRAMEWORK TO DEVELOP EMBEDDED SOFTWARE USING ASYNCHRONOUS UNITS

IoT devices commonly operate using an event-driven model, where actions and responses are determined by inputs and outputs (I/O) (KIM; JEONG; MOON, 2017). Consequently, IoT devices are well-suited for event-driven programming, allowing embedded programmers to develop applications following this paradigm. Therefore, coding programs using an event-driven model is a natural and fitting choice for IoT devices.

This chapter introduces a framework that assists developers in building embedded software from an event-driven perspective, allowing actions to be performed asynchronously. Furthermore, this approach is based on the JavaScript runtime execution pipeline.

5.1 JSEVAsync Proposal

Commonly, on resource-constrained devices, the programs follow the Time-Triggered (TT) architecture, in which algorithms are invoked periodically at specific intervals (NAHAS; NAHHAS, 2012). In practice, the programs are structured in two sections i) initialization/configuration and ii) while loop (polling); usually, they are written using the C language (EBERT; JONES, 2009). Although this approach can be seen as simple, the system always operates at full power, which might lead to significant energy consumption (NAHAS; NAHHAS, 2012).

In contrast to the TT approach, the Event-Triggered (ET) architecture proposes modeling algorithms for response events (NAHAS; NAHHAS, 2012). The events are unpredictable and aperiodic. For instance, a click event of a physical button represents a situation in which it is impossible to know when the event will happen. Nevertheless, the arrival of several events at the same time can cause overload and crashes (SCHELER; SCHROEDER-PREIKSCHAT, 2006).

Considering the countless forms for architecting and developing embedded software, we define the hypothesis that an alternative design that considers the main characteristic of the target environment, specifically events for IoT, might be more energy-

efficient if integrated with a suitable programming language, even if it was not originally designed for that environment.

Therefore, we propose a strategy that combines time- and event-triggered architectures to build optimized applications focusing on energy saving. Energy consumption has become a key issue in embedded systems (GERHORST et al., 2020) and considering a context based on events such as the Internet of Things, it would be natural to develop applications in an event-driven model.

This section introduces JSEVAsync, a framework to help developers design applications for IoT devices using the JavaScript language that combines the best parts of the TT and ET strategies. This approach uses JavaScript's non-blocking concept as a development interface to structure algorithms into asynchronous events.

JSEVAsync was designed as a discrete event-based model of computation. The framework generates and processes events over time using a JavaScript virtual machine. We assume that functional requirements, I/O operations, and communication actions could occur through events. Thus, the design of the application should follow an event-triggered architecture. However, the framework must also be able to handle multiple events. To address this, we introduce a time-triggered approach to ensure correct event handling. It means we leverage the single-threaded nature of the call stack, along with the combination of the callback queue and event loop mechanism, to manage and handle multiple events. Figure 38 provides an overview of this approach.

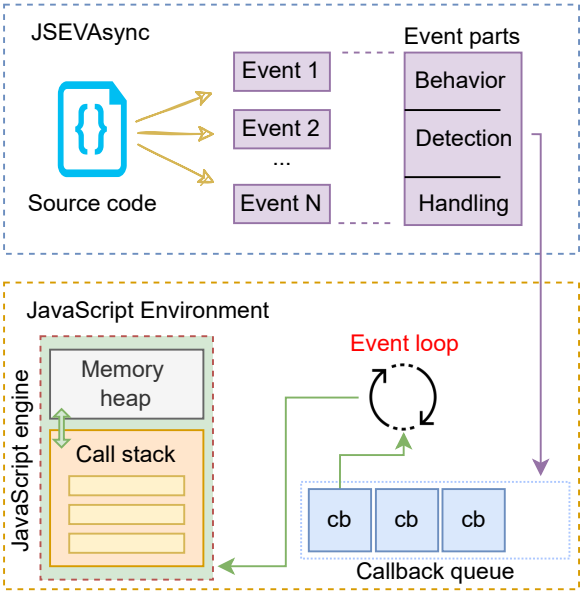


Figure 38 – Overview of JSEVAsync.

Figure 38 illustrates the basic conceptual architecture of the JSEVAsync framework and its integration with the JavaScript runtime environment. The system behavior is modeled over an event-driven perspective. Therefore, algorithms must be designed to split their concerns into fragments that can be executed and monitored individually.

Indeed, events are transformed into promises to be executed in the future. Once a promise is resolved (finished), its result is associated with a function (callback function) responsible for handling the result. Finally, the event loop pulls a callback into the call stack to be processed.

The JavaScript runtime environment facilitates the integration and scheduling of all events. Its conceptual principle of being single-threaded ensures that only one event can be handled at a time. Based on this, events can be queued and processed through the JS execution pipeline, optimizing the use of the CPU and saving energy.

Overall, the execution of the application follows the JavaScript runtime flow. This methodology makes the ET architecture quite attractive because its execution model allows us to create multiple events to implement functional requirements and delegate their control to the JS virtual machine. In contrast, the single-thread strategy adopted for the call stack is compatible with the TT strategy, ensuring that only one event is handled at a time and supporting the handling of multiple events (the available hardware resources determine the limit). This helps to ensure the predictability of the algorithms, which is an important design requirement for embedded software (NAHAS; NAHHAS, 2012).

Once an event is detected, it needs to be handled. According to the previous discussion, the event is handled asynchronously. Thus, a callback function will be queued to be processed in the call stack. In other words, the event loop fetches the events, executes them, and may register new events into the queue.

For instance, suppose there is a sensor whose data needs to be sent to the server periodically. The initial event registers a function to read the sensor. When the operation is completed, a trigger is fired containing the result. Finally, another callback is registered to send the data over the network.

In order to use JSEVAsync, the concerns of the application need to be mapped into events. Inspired by Rahman; Ozcelebi; Lukkien (2018)'s work, we created a set of four events that represent common situations in IoT applications, namely: a) events that repeat continuously, b) one-shot execution, c) events fired on demand, and d) functions that repeat after a certain period. Figure 39 exposes the events type.

Each event can be used to cover specific system requirements, and its execution (behavior) is determined according to the event's type. Thus, the developer must choose which event best fits the application logic. In the following, we detail the purpose of each kind of event mapped.

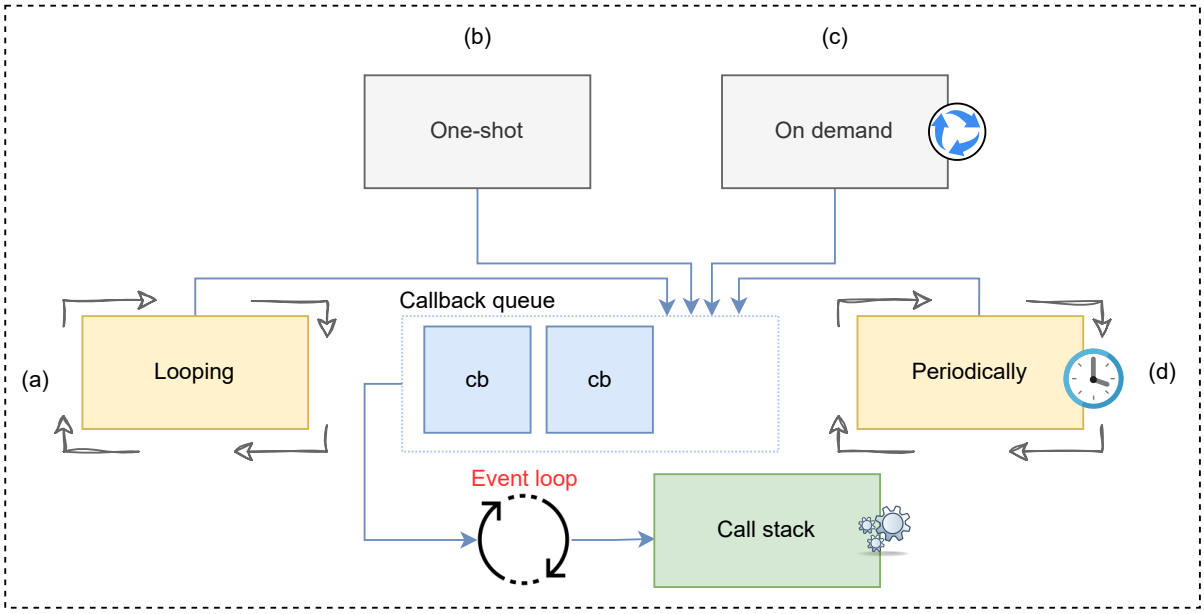


Figure 39 – Type of events.

- **Looping:** Represents a classic super loop logic. This event runs indefinitely without delays, with each interaction encapsulated inside a promise.
- **One-shot:** Equivalent to a situation where the execution is required only once, such as setting up hardware.
- **On demand:** This event represents a situation where actions must be performed several times but not in constant or cyclic situations, such as sending data to the server over the network. The task may not be deterministic, depending on the application logic;
- **Periodically:** Similar to a looping event, but with a specified time interval in milliseconds (delay) between each interaction.

To validate JSEVAsync, we applied the framework to build an IoT application. To measure the effectiveness of the solution, we created two versions of the same application - one using JavaScript and the other using the C language - and compared their results. The application is an alarm system that monitors the environment and notifies the user when motion is detected. Figure 40 provides an overview of the proposed application.

Figure 40 demonstrates the requirements of the developed application. This scenario requires the system to monitor the environment continuously, and if a violation occurs, the device should notify the application server. Communication between the device and server demands extra power; therefore, we simulated some violations per test. Furthermore, each algorithm was deployed individually on the device, and the

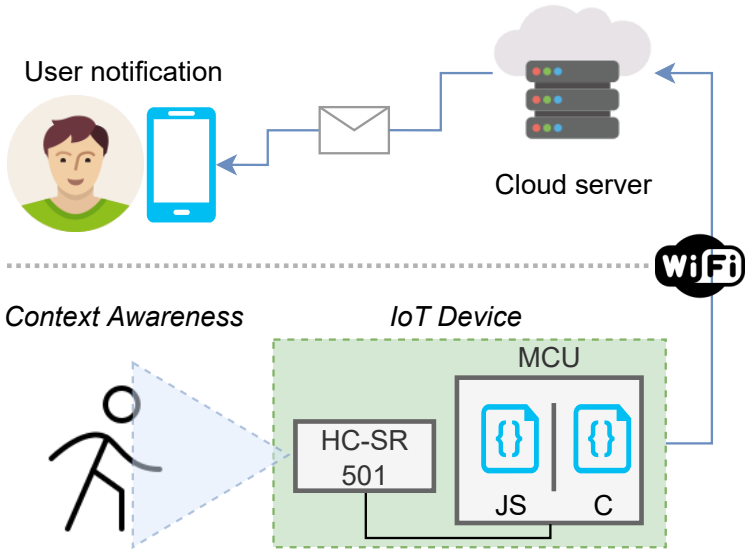


Figure 40 – Alarm System: Application flow.

data extraction was performed for eight minutes. Eight minutes is the limit of the ampere meter, considering the maximum sampling rate (100k per second). The server side of this application will not be covered in this investigation, kipping the focus on the embedded software. Figure 41 illustrates the experimental setup.

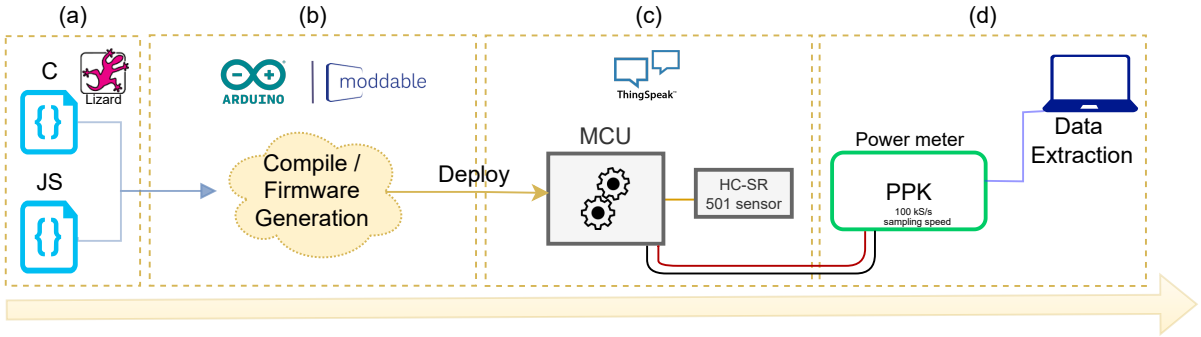


Figure 41 – Alarm System: Experimental setup.

Figure 41 depicts an overview of the experiment setup. In section (a), the source code is developed to maintain the same behavior in both algorithms. Furthermore, this section represents the execution of static code analysis to obtain software quality metrics. For example, we use the Lizard tool (TERRY YIN, 2023) to collect the Cyclomatic Complexity metric. Next, section (b) represents the code compilation phase and firmware packaging. The C code is compiled using the Arduino IDE, while the JavaScript program is processed, optimized, and compiled using the Moddable toolchain. Subsequently, the application is sent to the device. Section (c) illustrates the device setup and the application's execution. We also emphasize the use of the ThingSpeak service (JUNG et al., 2021) as a cloud server. Finally, section (d) describes the data collection and reporting of energy consumption.

5.2 JSEVAsync Validation

JSEVAsync introduces a novel approach to model algorithms as asynchronous events. The key concept is to leverage the optimized consumption of resources promoted by the event-driven strategy mixed with the JavaScript runtime to ensure proper execution of the applications. Therefore, developers must split the application requirements into events and handle them as promises, which will be managed within the JavaScript runtime pipeline.

As a proof of concept, we created an Alarm System using JSEVAsync and C language and compared the results. Specifically, the proposed experiment required periodic readings from a motion sensor. If a motion is detected, a local siren should ring, and a message must be sent to the server. We chose the ESP32 and Raspbery Pi Pico W as the MCU, Moddable SDK as the JavaScript engine and the HC-SR 501 sensor (COMPONENTS INFO, 2023b) for motion detection. Listing 5.1 provides the key parts of the C algorithm.

```
1  int sensorPin = 5;
2  // ...
3 void setup() { ... }
4
5 int sendDataToServer(float status) {
6   HTTPClient http;
7   String url = "thingspeak.com/<HIDE>&field1=" + String(status);
8   http.begin(url.c_str());
9   return http.GET();
10 }
11
12 void loop() {
13   if ( WiFi.status() == WL_CONNECTED ) {
14     int val = digitalRead(sensorPin);
15     digitalWrite(sirenPin, val);
16     if (val == HIGH) {
17       sendDataToServer(val);
18     }
19     //delay(1000);
20   }
21 }
```

Listing 5.1 – Alarm System implementation using C language.

Listing 5.1 represents the C implementation of the Alarm System. Some code fragments (configuration and underlying functions) were removed to focus on the central part of the code. This algorithm is a classic example of a super loop (NAHAS; NAH-

HAS, 2012). Although it sounds simple, its execution may consume more power because it always runs at full capacity. Moreover, it should be noted that the invocation of the method “SendDataToServer” (line 17) could block the program because this method performs actions that can be time-consuming and use I/O resources. Listing 5.2 shows the JavaScript version of the algorithm.

```
1 import { JSEVAsync } from "jsevasync";
2
3 const app = new JSEVAsync();
4 app.setup( ... );
5
6 app.addTask("sendToServer", (data) => {
7   let request = new Request({
8     host: "api.thingspeak.com",
9     path: `/<HIDE>&field1=${data.value}`,
10  });
11 });
12
13 app.createMonitor({
14   pin: 5,
15   onChange: (context) => {
16     const value = this.read();
17     Digital.write(context.ledPin, !value);
18     app.invokeTask("sendToServer", { value });
19   },
20 });
```

Listing 5.2 – Alarm System implementation using JavaScript language.

Listing 5.2 represents the algorithm modeled through the JSEVAsync framework. First, the program was coded following the object-oriented notation. The main object, “app,” (line 3) refers to an instance of the JSEVAsync class, and from this reference, the program is organized to meet the application requirements.

Except for the setup method (which has been removed), Listing 5.2 presents the rest of the JavaScript program without deletions, demonstrating that the code is compact, which simplifies understanding and future maintenance interventions.

In particular, we created a monitor to read the motion sensor signal (line 13). The Monitor is a special class from the XS engine that can act as a listener for changes in digital ports; in this case, port number 5. A monitor can detect changes in digital input value (from 0 to 1 or 1 to 0) and trigger a specific callback.

The callback function (lines 15-19) was created to handle the event, and in line 18, there is a nested call to the “sendToServer” event. This call represents the execution

of a chunk of code to perform an I/O operation that will be done asynchronously. In other words, we want to run the function, but we do not know when it will occur. Thus, this instruction represents the intention to act, which becomes a promise. The event loop manages and executes the promise on the call stack, thereby preventing program blocking.

In line 6, we created a promise to perform a request to the cloud server. This task can be triggered manually and invoked several times. In addition, the method “addTask” returns an object of the type Promise. Hence, it will only be processed if there is a handler for success (resolve) and optionally for error (reject). This method also allows task customization to perform repetitions and delays between executions. Listing 5.3 presents a simple example of a polling operation developed using JSEVAsync.

```
1 import { JSEVAsync } from "jsevasync";
2 const app = new JSEVAsync();
3
4 app.addTask("blink", (data) => {
5     data.value = !data.value;
6     Digital.write(2, data.value);
7  },
8  {
9     repeat: true,
10    delay: 1000
11 }
12 });
```

Listing 5.3 – JavaScript blink example created using JSEVAsync.

Listing 5.3 illustrates an implementation example of a task that repeats indefinitely. Specifically, this snippet represents an infinite LED blink; line 9 sets the loop mode, and line 10 determines the delay (in milliseconds) between executions. This algorithm is interesting because it demonstrates how to model a program using a polling strategy with JSEvAsync (periodic event type). Behind the code, JavaScript can check whether the event loop has anything to do, and if not, it can enter idle mode and consequently reduce the frequency of operation, thereby saving energy.

5.2.1 Code Quality Analysis

Regarding the design-time metrics of the source code, using the JavaScript language to code embedded software can increase the abstraction level and improve readability, maintainability, and reusability of the source code. Figure 42 lists the code quality metrics.

In an empirical analysis comparing algorithms, a programmer can classify them as equivalent in terms of complexity. However, this perception is based on the number

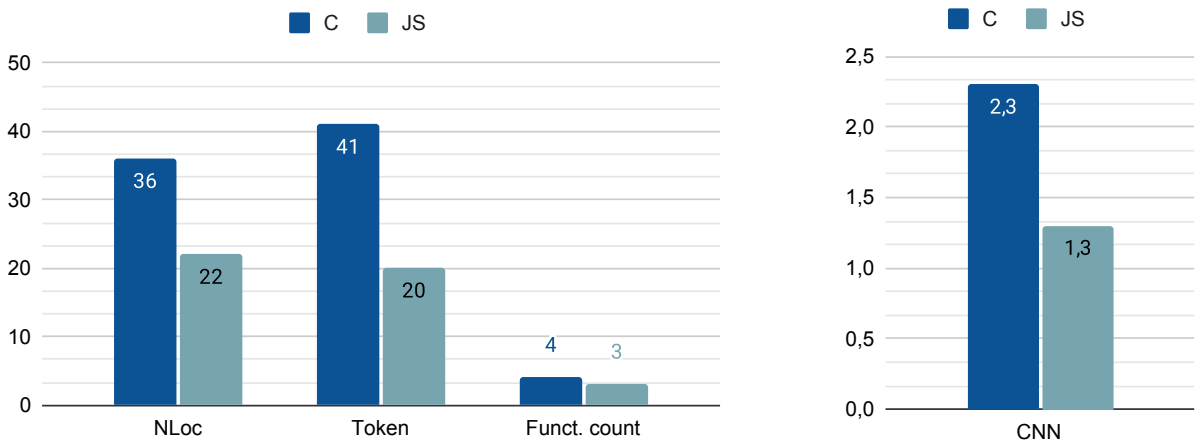


Figure 42 – Alarm System: cyclomatic complexity analysis.

of code lines, which may not be an adequate metric, especially when comparing distinct languages. Thus, the use of tools to measure the complexity level of programs is essential. Moreover, the use of cyclomatic complexity as a design and development guideline can lead to more efficient and reliable software for embedded systems (EBERT et al., 2016). Table 11 shows the Halstead metrics extracted from the Alarm System algorithms.

Table 11 – Alarm System: Halstead metrics.

Metric	C	JavaScript
Program length	382	270
Program vocabulary	66	51
Estimated length	340.52	250.93
Purity ratio	0.89	0.93
Volume	2308.96	1531.55
Difficulty	56.52	26.12
Program effort	130,506.35	40,011.87
Time to program (h)	2.10	1.01

The Halstead metrics achieve similar results to previous analysis. JavaScript, in terms of code quality, performed better than C code. Thus, this reinforces the notion that interpreted languages can have a simpler syntax that favors activities related to post-development.

5.2.2 Resource Consumption Analysis

To measure resource consumption, we emulated violations to force data transfer to the server. These violations were established to occur every 20 seconds, resulting in a total of 25 transfers during the experiment.

Energy consumption represents all the energy expended by the microcontroller since its initialization, connection to the Wi-Fi network, and data transfer. Table 12

provides a detailed comparison between the algorithms.

Table 12 – Alarm System: Energy consumption by language.

Language	Avg. Curr. (mA)	Max Curr. (mA)	Energy (mJ)
C	90.77	550	12.61
JS	69.21	580	9.68
%	-31.15	3.85	-30.18

Table 12 presents the comparison between the algorithms. JavaScript is considered better than C when the percentage value is negative (i.e., it reduces energy consumption), and vice versa when it is positive. The JS algorithm could save 30.18% of the energy required for this experiment, which required constant monitoring of the environment.

To understand the gain, we need to reflect on the Time-triggered architecture. From Listing 5.1, the loop method promotes intense CPU usage. In particular, the code performs reading and writing to a digital port to check for changes. In contrast, the JavaScript strategy uses the ET approach. The event is triggered based on built-in hardware that detects the change and generates an interrupt.

Naturally, a hardware interrupt would be more efficient than a polling strategy (software). However, if this feature is native to the microcontroller, could it be implemented in the C algorithm? The answer is yes. However, implementing a listener in a non-event-driven language is complex and requires a high level of domain knowledge. Moreover, it increases the code volume, negatively impacts maintenance, makes reuse difficult, and increases the overall footprint. Nevertheless, in order to compare the development styles, we proceeded with changes to the C algorithm to use interrupts instead of polling. Listing 5.4 show part of the algorithm implementation and Table 13 presents the results.

```
1  int sensorPin = 5;
2  // ...
3 void setup() {
4  ...
5  attachInterrupt(digitalPinToInterrupt(sensorPin),
     checkSensorData, CHANGE);
6 }
7
8 void checkSensorData(){
9  if ( WiFi.status() == WL_CONNECTED ) {
10     int val = digitalRead(sensorPin);
11     digitalWrite(sirenPin, val);
12     if (val == HIGH) {
```

```
13        sendDataToServer(val);
14      }
15   }
16 }
17
18 int sendDataToServer(float status) {
19   HTTPClient http;
20   String url = "thingspeak.com/<HIDE>&field1=" + String(status);
21   http.begin(url.c_str());
22   return http.GET();
23 }
24
25 void loop() {}
```

Listing 5.4 – Alarm System implementation using C language and interrupt approach.

The algorithm using interrupts may give the impression of simplicity; however, this perception arises because there are no actions running in the loop method. If other routines were present, the interrupt would require additional controls to manage the flow of execution between the interrupt and the loop. Thus, this can make the maintenance and evolution of applications more challenging.

Table 13 – Alarm System: Energy consumption by language using hardware interrupt.

Language	Avg. Current (mA)	Max Current (mA)	Energy (mJ)
C	71.44	540	10.02
JS	69.21	580	9.68
%	-3.12		-3.39

JavaScript still saves energy, but the difference is minimal. The interrupt implementation in the C language was based on the native method called “attachInterrupt,” using variables to control events and triggers. Overall, the interrupt strategy in both scenarios can save energy compared to the polling strategy. However, the analysis demonstrated that the Monitor class from Moddable can be an even more efficient solution.

To verify whether excessive consumption by the C algorithm is determined by the number of readings and writings, we introduced a delay function (specifically on line 19 of Listing 5.1). We then repeated the tests, and the results are listed in Table 14. In adition, Figure 43 show the results comparing all the implementations.

As shown in Table 14, JavaScript remained the most energy efficient. It is important to note that the delay was inserted only in Algorithm C. The delay function helps reduce the energy consumption. Therefore, can we conclude that this function saves energy? No, introducing the delay instruction minimizes the number of sensor readings and writings, and this action effectively reduces consumption. However, when the

Table 14 – Alarm System: Energy consumption by language with delay.

Language	Avg. Current (mA)	Max Current (mA)	Energy (mJ)
C	72.03	540	10.07
JS	69.21	580	9.68
%	-4.07		-4.02

algorithm is in a delayed state, the code still runs to check the remaining delay time, and therefore, still consumes energy.

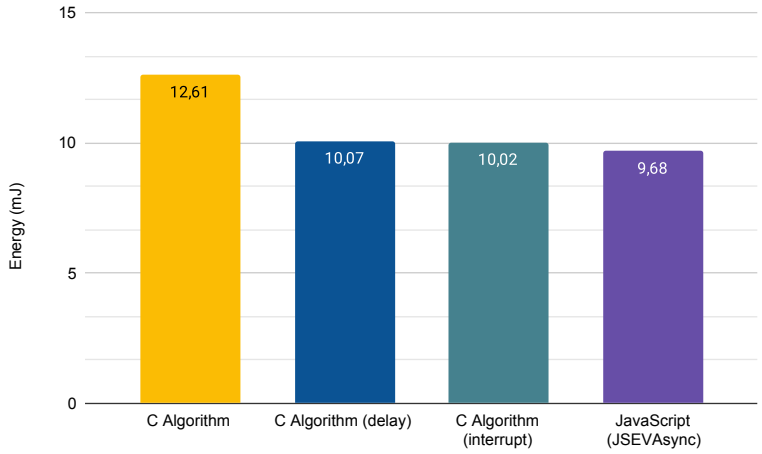


Figure 43 – Alarm System: Energy consumption by implementation.

Regarding memory consumption, the ESP32 has 520 kB of RAM, with 200 kB reserved for instructions and 320 kB available for application use. Table 15 displays the amount of memory used by each language.

Table 15 – Alarm System: Memory consumption (kB).

Language	Memory used	Free memory	Available memory	Footprint
C	61.90	258.84	320.75	834.93
JS	65.12	100.76	165.87	919.46
%	4.93			9.19

Memory consumption was balanced, with a difference of only 4.95% in favor of the C language. This measurement evaluates the consumption of the entire device, including the overhead of the JavaScript virtual machine. In addition, the total amount of memory differs between languages, with the C language utilizing all available space, whereas JavaScript maps a smaller amount of memory. Mapping less memory allows for smaller indices in the indexing tables that map data from ROM to RAM, which is XS's strategy for saving RAM. However, the specific consumption of the algorithm was minimal. By default, XS sets a limit of 32 kB for application use, and in this example, the algorithm consumed only 812 bytes from this limit.

Given the nature of this algorithm, which reacts based on the value read from a sensor, we also assessed the time taken to perform the effective reading of the sensor and send data over the network. However, this measure is not precise because the overhead of the Internet can influence the results. Nonetheless, it provided interesting insights. The average time for Algorithm C was 1430 ms, while JavaScript took 282 ms. These numbers highlight a significant difference, which occurs because JavaScript does not wait until the networking operation finishes; thus, the time basically represents the sensor reading. However, it is possible to wait for the end of the data transfer, in which case the average time for JavaScript was 2320 ms.

Finally, the results reaffirm that the event-driven approach is energy-efficient. However, the execution time and memory consumption are still not as good as those of compiled languages. Nevertheless, they are close, and considering the advantages of working with an interpreted language, this difference may not be significant. Of course, it depends on the characteristics of the application, such as response time, reliability, size, cost and hardware. Therefore, JavaScript can be a viable alternative for IoT applications.

5.3 Related work

Developing an embedded system is not a trivial task. Therefore, we have selected works that focus on Event-triggered (ET) and Time-triggered (TT) approaches related to the Internet of Things (IoT) or provide comparisons between them.

Ayestaran et al. (2014) present a platform-specific time-triggered model (PS-TTM), which is a systems-based modeling approach founded on time-triggered architecture. This approach facilitates the modeling of Time-Triggered Architecture for embedded systems. The authors validate their framework through a case study that involves modeling, simulation, and validation of a simplified railway onboard signaling system.

In a comparative study between Event-triggered and Time-triggered architectures, Thaker; Babu (2017) explore the main features of each approach, implement a safety-critical real-time prototype, and propose a TT design pattern using a Real-Time Operating System (RTOS).

In their research, Wang et al. (2017) explore asynchronous events for establishing communication between virtual machines in the context of embedded systems. They introduce a model for asynchronous communication using OKL4 technology. The proposed mechanism allows concurrent communication by adopting event channels, making their experiment feasible and effective.

Addressing the event triggering approach for IoT applications, Kolios et al. (2016) characterize the Event-triggered (ET) architecture in the IoT domain, considering behavior modeling, event detection, and event handling. As a result, the authors propose

a data-driven technique and compare it with existing periodic methods.

Given the context, it is worth noting that researchers have been investigating the use of both TT and ET architectures in embedded systems. Thus, this study distinguishes itself by providing an approach that combines ET and TT architecture styles, leveraging the features of the JavaScript language, with a specific focus on energy efficiency.

5.4 Summary

This chapter introduces JSEVAsync, a framework that helps developers build embedded software from an event-driven perspective using the JavaScript language. The framework combines the time-triggered and event-triggered architectures to optimize energy consumption in IoT applications.

Furthermore, JSEVAsync uses JavaScript's non-blocking concept and asynchronous event handling. The framework conduct the developers to model software inspired by the JavaScript runtime execution model. Moreover, the text also presents a validation of JSEVAsync by comparing it with a C language implementation in an IoT alarm system application. The JavaScript implementation using JSEVAsync is more compact and provides asynchronous execution, improving energy efficiency.

6 DISCUSSIONS

This chapter is dedicated to discussions and reflections about the research results, serving as a critical component of the thesis. In addition, we thoroughly examine and provide a comprehensive analysis and interpretation of the research findings. Finally, we explore the implications and significance of the results of the research objectives and questions.

Developing applications for heterogeneous contexts like IoT, while challenging, requires a simple programming model to be feasible (KRISHNAMURTHY; MAHESWARAN, 2016). Also, the advancements in virtual machines such as Espruino and XS make JS a viable option for programming constrained devices.

We started our experiments using the Espruino engine because it was the first engine found in the literature review that met our expectations of running JavaScript on resource-constrained devices. However, during the literature review update, we came across a paper by Grunert (2020) that presented the XS engine. Henceforward, we started to use the XS engine as our official virtual machine to validate JavaScript in the IoT context.

Our experiments have demonstrated the practical superiority of the XS engine. It excels by consuming fewer resources, resulting in lower overhead. Moreover, XS distinguishes itself by providing extensive coverage of the latest JavaScript API, making it a comprehensive and feature-rich development environment. It offers a robust set of tools, libraries, and APIs that enable developers to build sophisticated applications with ease.

One of the most valuable advantages of using JavaScript to code embedded software is the possibility of abstracting architectures and providing generic access to different technologies. For example, when we use the C language to obtain technical performance data, such as the amount of available memory, we must use the API provided by the microcontroller manufacturer. In the case of ESP8266 and ESP32, this is the ESP-IDF API. However, JavaScript can abstract this control and make it transparent to developers; therefore, they only need to know the performance API.

The quality of the software is crucial for embedded systems, especially when ap-

plied to an interpreted language, as it can be decisive for its adoption. Furthermore, considering that there is a direct relationship between the quality of the code, resource consumption, and performance, it is essential to build software with consideration for the best resource consumption (OLIVEIRA et al., 2008a; PAPADOPOULOS et al., 2018). Thus, the guidelines produced can be useful in this regard.

In this context, code smells can be used as indicators of source code quality, meaning that low-quality codes can directly affect maintenance efforts (YAMASHITA, 2013; BOGNER; MERKEL, 2022). At the same time, detecting and removing code smells can prevent errors (EMDEN; MOONEN, 2012). Therefore, we can affirm that the results of code smell analyzes can be used to suggest a guide for best practices to improve code quality in an embedded software context. Furthermore, its adoption can have an impact on resource consumption and mainly contribute for the design-time metrics.

Nonetheless, it should be noted that our approach to detecting code smells considers only static code analysis and therefore corresponds to only a part of the development process or application execution cycle. Indeed, the behavior of the code should also be evaluated through dynamic analysis. However, considering our study scenario, which involves an embedded context, such analysis becomes more complex because algorithms may interact with sensors, actuators, networks, and other elements that are not available at the time of design. Thus, evaluating the code from this perspective remains a challenge in the context of embedded systems and may represent possible future work efforts.

Another potential improvement that we envision for JSGuide is the inclusion of estimated values regarding the cost of adopting specific techniques/recommendation. In addition to providing a description of the reported “problem” or situation, the tool could estimate the impact in terms of memory consumption, energy usage, or execution time. Providing concrete figures, such as stating that a certain piece of code can increase memory consumption by X%, would be more impactful than simply mentioning that it may consume more memory. This enhancement would provide developers with more valuable insights into the potential consequences of adopting certain coding practices.

After conducting studies and investigations, a question that may still be unclear is: How can JavaScript save more energy than C, considering that JS is an interpreted language? To answer this question, we need to understand the JavaScript execution model proposed by Moddable.

Considering that constrained devices typically have less RAM than ROM, with some devices having only a few kilobytes of RAM while ROM can be in the order of megabytes, Moddable adopts a strategy that prioritizes executing ECMAScript code from ROM rather than RAM. Moreover, the XS engine implements various techniques for reducing the required ROM footprint. As a result, when the application starts, the embedded device boots instantaneously as everything is already prepared. Further-

more, since nothing is copied from ROM to RAM, the application runs efficiently in just a few kilobytes of memory.

Some studies, such as Wolf; Kandemir (2003); Verma; Marwedel (2007); Oliveira; Mattos; Brisolara (2013); Hennessy; Patterson (2017), have suggested that excessive power consumption may be attributed to memory usage. Therefore, reducing the memory usage can contribute to energy savings. In addition, the XS engine minimizes the amount of RAM required, which can result in lower power consumption.

Furthermore, JavaScript improves the design-time aspects of algorithms, such as readability, code reuse, and maintainability. Hence, JS can increase developer productivity due to its platform-independent syntax and higher level of abstraction.

Life is not a bed of roses, and JavaScript is no exception. JavaScript has its drawbacks, commonly associated with the concept known as “JavaScript hell.” This term refers to the challenges and complexities that developers may encounter when working with JavaScript, especially in large and complex codebases (TAIVALSAARI; MIKKONEN, 2017).

One aspect of JavaScript hell is its loosely typed nature, which can lead to unexpected behavior and bugs. Without the strict type checking found in other languages, it becomes easier to introduce errors during development and more difficult to identify them. This can result in time-consuming debugging and troubleshooting processes. To mitigate the negative impact of JavaScript hell and improve the development experience, developers can use tools and frameworks to assist themselves. In this regard, JSEVAsync can be a useful tool to enhance the development experience.

JSEVAsync provides a powerful framework that addresses some of the challenges associated with JavaScript’s asynchronous nature, particularly in handling callbacks and managing asynchronous operations. This framework introduces a more streamlined and structured approach for managing asynchronous tasks.

One of the key features of JSEVAsync is its utilization of promises. Promises provide a way to handle asynchronous operations and simplify the flow of code. JSEVAsync leverages promises to manage callbacks and handle asynchronous tasks in a more organized and readable manner. By leveraging the JavaScript runtime model, developers can chain asynchronous operations together and handle errors more effectively, resulting in cleaner and more maintainable code.

In our case study, we used an alarm system as the basis for comparison, where we implemented the business rules using a polling algorithm in C. While this may not be the most appropriate comparison, polling is commonly used as a reference for documentation on interactions between components that communicate through digital ports. For example, in Arduino’s reference base, polling is presented as the default approach.

However, when we applied interrupts in the C algorithm, we reduced the difference in power consumption. However, additional variable declarations were necessary

to control the application state and ensure the correct functioning of the algorithm. Therefore, we reiterate that JavaScript provides a standardized development approach without the need to create additional structures to implement business rules.

However, it is important to highlight that we have currently mapped four types of events for JSEVAsync, which align well with our experiment. However, for other real-world applications, new event types may need to be created. Therefore, the framework still requires further development to cover a broader range of scenarios, including different sensors and actuators, while respecting their specificities and requirements.

In previous work (OLIVEIRA; PARIZI; MATTOS, 2022), we discovered a scenario in which JavaScript did not perform well, although it proved to be efficient in executing algorithms. In a scenario with limited battery power, strategies are adopted to reduce energy consumption, with deep sleep being particularly notable. However, we identified that JavaScript consumes much more time and energy to initialize and reconnect to the Wi-Fi network when the device returns from deep sleep mode. Therefore, the language may not be the most suitable choice for this specific scenario. However, it is worth noting that it is possible to utilize native functions for specific situations. Thus, it is possible to devise a mechanism to minimize the adverse effects of this problem and maintain JavaScript as a viable alternative.

Finally, one of the key benefits of using JavaScript on resource-constrained devices is its ease of use and familiar syntax for developers who are already familiar with Web development. This allows for the faster development and prototyping of applications for IoT devices. However, some libraries used by developers may not be compatible with embedded devices, either because they require a lot of resources or because they were built based on components that are not available on the devices. For instance, components that use scheduling resources native to the Node.js platform are unavailable for embedded development.

7 CONCLUSIONS AND FUTURE WORK

This study analyzed the use of an interpreted language as an alternative to the C language for coding embedded software for IoT devices, particularly on resource-constrained devices. We proposed improvements to reduce the performance gap between compiled and interpreted languages, and developed a framework to help embedded programmers model software through asynchronous units.

We initiated the discussion assuming that interpreted languages could bring any benefits to embedded software development. After conducting research and experimentation, we were able to measure the potential of the JavaScript language from both design-time and runtime perspectives.

Therefore, answering the research question: *Can an interpreted language be used to develop high-quality embedded software for devices with limited resources?* - Yes, it is feasible. However, there are some reservations about this topic.

First, IoT solutions are not as simple as we initially proposed. Nevertheless, the examples provided represents common scenarios for IoT applications. Furthermore, due to the hardware restrictions of many IoT devices, the majority of applications are not critical and usually involve simple activities such as sensing.

For those applications that are critical, JS is still an alternative. To address performance-critical code, the JavaScript, through the XS engine, allows for native code invocation. Therefore, working with JS does not necessarily imply compromising – native code is always an option. The compromise is choosing to work only in a compiled language like C, where the benefits of JavaScript cannot be available.

our experiment demonstrated that JavaScript could enhance code reuse and enable the building of standard components. For instance, in automation systems that need to consider various hardware sensors and actuators, JavaScript provides the basic infrastructure necessary to create solutions, frameworks, and libraries to integrate all of them, thereby improving the developer experience.

Second, we can conclude that performance is strongly related to code quality and the virtual machine. While the algorithm itself has a significant contribution to how the application performs, it is the VM that translates it to machine language. Therefore, the

application's performance depends on how optimized the engines are. Thus, programming for IoT devices is not a trivial task, and we need to find a balance between the development, execution, and resource consumption.

In this study, we used two virtual machines: Espruino and Moddable XS. Moddable XS is the preferred option due to its superior performance and coverage of ECMA specifications. While Espruino is a useful option for embedded development, it can add a significant overhead, and its ECMA conformance may limit algorithmic capabilities.

Interpreted languages like JavaScript can be used to develop embedded software, as long as they are supported by appropriate tools that assist in development and guide programmers in creating solutions suitable for the specific context in which they will be applied. This thesis contributes two tools: JSGuide and JSEVAsync.

Although JSGuide is a helpful tool for developers to evaluate code quality and improve their coding skills, it is up to the developer to decide whether to adopt the suggestions offered by JSGuide and refactor their code. Moreover, the knowledge provided by JSGuide can help reduce technical gaps caused by a lack of experience and aid in the decision-making process to improve the algorithm's efficiency.

Regarding energy consumption, the event-driven approach proves to be suitable for saving energy. The JSEVAsync framework proposes a mechanism that combines the usual time-triggered style adopted in real-time systems with the event-triggered approach. As a result, algorithms can be written to be more energy-efficient. Moreover, code readability is improved by adopting JSON notation to describe the event, and the object-oriented paradigm enhances the algorithm, making it more intuitive and clear.

Introducing interpreted languages to the IoT context represents the unification of the programming language, enabling the use of the same programming language across the web, server, mobile, and IoT. It simplifies and standardizes the development of applications, positively implying a more favorable environment for interoperability and becoming the basis of the cross-platform development environment.

Finally, for all that, enabling high-level language to code IoT devices can contribute to programmable capabilities, increase abstraction level, and code reuse. Consequently, it can be helpful to attend to the software requirements. Therefore, the JavaScript language can be suggest as good alternative to code resource-contrained devices.

7.1 Future Work

The interpreted languages have been used in several domains. However, most of them were not designed for embedded contexts. Therefore, their structures, API, and underlying tools may not be optimized for use in limited environments. Accordingly, there are many open questions regarding the use of interpreted languages in the IoT domain, especially for resource-constrained devices. Thus, we propose the following directions for future investigation:

- **Exploring new use cases:** As JavaScript has proven to be feasible on constrained devices, we could suggest, as future work, exploring new use cases and developing new applications that leverage the unique features of constrained devices and the diversity of embedded software requirements.
- **Security and privacy:** Security and privacy are always important considerations when developing software, but they become even more critical when working with embedded systems that may have to cope with personal data. Thus, we could suggest developing/studying techniques to guarantee the security and integrity of users' data that should be transferred between application layers.
- **Dynamic code analysis:** Exploring the use of dynamic code analysis techniques, such as runtime profiling and tracing, to capture execution information such as method invocations, parameter values, and data flow. This information can be used to identify patterns or anomalies that may indicate the presence of code smells.
- **Refactoring**: Based on JSGuide results, which provide valuable suggestions for code improvements, we could suggest the development of a tool that can analyze the source code and apply appropriate refactoring techniques based on detected code smells. This tool would assist developers in automatically refactoring their code, simplifying the process and ensuring consistent and effective improvements.

7.2 Publications

Throughout the research development, we have been concerned about publishing the partial work's results so that other researchers could evaluate and contribute to our research. Thus, the following papers were published during the doctoral program:

- State-of-the-Art Javascript Language for Internet of Things. In IX Brazilian Symposium on Computing Systems Engineering (SBESC) (OLIVEIRA; MATTOS, 2019)
- Webassembly: Uma Estratégia Para Melhorar O Desempenho Da Aplicação JavaScript Em Ambientes IoT. In Anais 6 Semana Integrada UFPel (SIIEPE) / Encontro de Pós-Graduação (XXII ENPOS) (OLIVEIRA; MATTOS, 2020b)
- Analysis of WebAssembly as a Strategy to Improve JavaScript Performance on IoT Environments. In X Brazilian Symposium on Computing Systems Engineering (SBESC) (OLIVEIRA; MATTOS, 2020c)
- Improving Developer Productivity on Internet of Things using JavaScript. In Proceedings of the 7th International Conference on Internet of Things, Big Data and Security (IoTBDS) (OLIVEIRA; PARIZI; MATTOS, 2022)
- Towards Event-driven Context: JavaScript an Energy-efficient Language for the Internet of Things. P.h.D Forum of Embedded System Week 2022 (ESWEEK, 2022)
- JSGuide: A Tool to Improve JavaScript Algorithms Focusing on IoT Devices. In Symposium on Internet of Things (SIoT) (OLIVEIRA; MATTOS, 2022a)
- JSEVAsync: An Asynchronous Event-based Framework to Energy Saving on IoT Devices. In XII Brazilian Symposium on Computing Systems Engineering (SBESC) (OLIVEIRA; MATTOS, 2022b)
- A Hybrid Approach to Design Embedded Software Using JavaScript's Non-blocking Principle. In Student Research Abstract of 38th Symposium on Applied Computing - (OLIVEIRA, 2023)

This work was awarded as one of the three best works in the Physical Sciences and Mathematics category at the Integrated Week of Innovation, Education, Research, and Extension (SIIEPE) 2020, promoted by the Federal University of Pelotas in Brazil. Furthermore, it was also considered as one of the four best works in the student abstract category at The 38th ACM/SIGAPP Symposium on Applied Computing, held in Tallinn, Estonia in 2023.

REFERENCES

The Computer Language Benchmarks Game . **The Computer Language Benchmarks Game**. Accressed: Mar. 2023, https://benchmarksgame-team.pages.debian.net/benchmarksgame/index.html.

ABDULSATTAR, K.; AL-OMARY, A. Pervasive Computing Paradigm: A Survey. In: INTERNATIONAL CONFERENCE ON DATA ANALYTICS FOR BUSINESS AND INDUSTRY: WAY TOWARDS A SUSTAINABLE ECONOMY (ICDABI), 2020., 2020. **Anais...** [S.l.: s.n.], 2020. p.1–5.

AL-FUQAHA, A. et al. Internet of Things: A Survey on Enabling Technologies, Protocols, and Applications. **IEEE Communications Surveys Tutorials**, [S.l.], v.17, n.4, p.2347–2376, 2015.

ALOSEEL, A.; HE, H.; SHAW, C.; KHAN, M. A. Analytical Review of Cybersecurity for Embedded Systems. **IEEE Access**, [S.l.], v.9, p.961–982, 2021.

ANDREASEN, E. et al. A Survey of Dynamic Analysis and Test Generation for JavaScript. **ACM Comput. Surv.**, New York, NY, USA, v.50, n.5, sep 2017.

Apple JavaScriptCore. **JavaScriptCore**: Apple Developer Documentation. Accessed: Mar. 2023, https://developer.apple.com/documentation/javascriptcore.

ASHTON, K. et al. That 'internet of things' thing. **RFID journal**, [S.l.], v.22, n.7, p.97–114, 2009.

ÅSRUD, M. **A Programming Language for the Internet of Things**. 2017. Dissertação (Mestrado em Ciência da Computação) — University of Oslo, Institutt for informatikk.

ATZORI, L.; IERA, A.; MORABITO, G. The internet of things: A survey. **Computer networks**, [S.l.], v.54, n.15, p.2787–2805, 2010.

AYESTARAN, I. et al. A novel modeling framework for time-triggered safety-critical embedded systems. In: FORUM ON SPECIFICATION AND DESIGN LANGUAGES (FDL), 2014., 2014. **Proceedings...** [S.l.: s.n.], 2014. v.978-2-9530504-9-3, p.1–8.

AZIZ, M. W.; ULLAH, N.; RASHID, M. A Process Model for Service-Oriented Development of Embedded Software Systems. **IT Professional**, [S.l.], v.23, n.5, p.44–49, 2021.

BABA-CHEIKH, Z. et al. A Preliminary Study of Open-Source IoT Development Frameworks. In: IEEE/ACM 42ND INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING WORKSHOPS, 2020, New York, NY, USA. **Proceedings...** Association for Computing Machinery, 2020. p.679–686. (ICSEW'20).

BACCELLI, E. et al. Reprogramming Low-end IoT Devices from the Cloud. In: CLOUDIFICATION OF THE INTERNET OF THINGS (CIOT), 2018., 2018. **Anais...** [S.l.: s.n.], 2018. p.1–6.

BAHETI, R.; GILL, H. Cyber-physical systems. **The impact of control technology**, [S.l.], v.12, n.1, p.161–166, 2011.

BAK, N.; CHANG, B.-M.; CHOI, K. Smart Block: A Visual Programming Environment for SmartThings. In: IEEE 42ND ANNUAL COMPUTER SOFTWARE AND APPLICATIONS CONFERENCE (COMPSAC), 2018., 2018. **Anais...** [S.l.: s.n.], 2018. v.2, p.32–37.

BARKALOV, A.; TITARENKO, L.; MAZURKIEWICZ, M. **Foundations of embedded systems**. [S.l.]: Springer, 2019.

BHATTACHARYA, P.; NEAMTIU, I. Assessing programming language impact on development and maintenance: A study on C and C++. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING (ICSE), 2011., 2011. **Anais...** [S.l.: s.n.], 2011. p.171–180.

BOGNER, J.; MERKEL, M. To Type or Not to Type? A Systematic Comparison of the Software Quality of JavaScript and TypeScript Applications on GitHub. In: IEEE/ACM 19TH INTERNATIONAL CONFERENCE ON MINING SOFTWARE REPOSITORIES (MSR), 2022., 2022. **Anais...** [S.l.: s.n.], 2022. p.658–669.

BRISOLARA, L.; MATTOS, J. **Desafios no Projeto de Sistemas Embarcados**. [S.l.: s.n.], 2009. p.153–175.

BYMA, S.; LARUS, J. R. Detailed heap profiling. In: ACM SIGPLAN INTERNATIONAL SYMPOSIUM ON MEMORY MANAGEMENT, 2018., 2018. **Proceedings...** [S.l.: s.n.], 2018. p.1–13.

CACCIAGRANO, D.; CULMONE, R. IRON: Reliable domain specific language for programming IoT devices. **Internet of Things**, [S.l.], v.9, p.100020, 2020.

CHO, S. Y.; DELGADO, R.; CHOI, B. W. Feasibility Study for a Python-Based Embedded Real-Time Control System. **Electronics**, [S.l.], v.12, n.6, p.1426, Mar. 2023.

CLAUSEN, L. R.; SCHULTZ, U. P.; CONSEL, C.; MULLER, G. Java Bytecode Compression for Low-End Embedded Systems. **ACM Trans. Program. Lang. Syst.**, New York, NY, USA, v.22, n.3, p.471–489, may 2000.

Components Info. **FS1000A 433MHZ RF transmitter**. Accessed: Mar. 2023, https://www.componentsinfo.com/fs1000a-433mhz-rf-transmitter-xy-mk-5v-receiver-module-explanation-pinout/.

Components Info. **HC-SR 501 sensor**. Accessed: Mar. 2023, https://www.componentsinfo.com/hc-sr501-module-pinout-datasheet.

CROCKFORD, D. **JavaScript**: The Good Parts: The Good Parts. [S.l.]: " O'Reilly Media, Inc.", 2008.

Duktape. **Duktape JavaScript Engine**. Accessed: Mar. 2023, https://duktape.org.

EBERT, C.; JONES, C. Embedded software: Facts, figures, and future. **Computer**, [S.l.], v.42, n.4, p.42–52, 2009.

EBERT, C. et al. Cyclomatic Complexity. **IEEE Software**, [S.l.], v.33, n.6, p.27–29, 2016.

Eclipse Foundation. **IoT Developer Survey 2020**. Accessed: May 2021, https://outreach.eclipse.foundation/eclipse-iot-developer-survey-2020.

Ecma International. **Standard definition of the ECMAScript 2021**. Accessed: Mar. 2023, https://www.ecma-international.org/publications-and-standards/standards/ecma-262.

ECMA International: Technical Committee 53 . **ECMA-419**: ECMAScript for Embedded Systems API specification. Accessed: Mar. 2023, https://419.ecma-international.org/.

EMDEN, E. van; MOONEN, L. Assuring software quality by code smell detection. In: WORKING CONFERENCE ON REVERSE ENGINEERING, 2012., 2012. **Anais...** [S.l.: s.n.], 2012. p.xix–xix.

ESP8266 NoDEMCU-12E. **Espressif Official Documentation**. Accessed: Mar. 2023, https://www.espressif.com/en/support/documents/technical-documents.

Espressif Systems. **ESP32 Series Datasheet**. Accessed: Mar. 2023, https://www.espressif.com/.

Espruino. **JavaScript Interpreter for Microcontrollers**. Accessed: Mar. 2023, https://www.espruino.com.

ETEROVIC, T. et al. An Internet of Things visual domain specific modeling language based on UML. In: XXV INTERNATIONAL CONFERENCE ON INFORMATION, COMMUNICATION AND AUTOMATION TECHNOLOGIES (ICAT), 2015., 2015. **Anais...** [S.l.: s.n.], 2015. p.1–5.

FABIAN, B.; MAURICE, A.; CHRISTIAN, B. A Mapping Language for IoT Device Descriptions. In: IEEE 43RD ANNUAL COMPUTER SOFTWARE AND APPLICATIONS CONFERENCE (COMPSAC), 2019., 2019. **Anais...** [S.l.: s.n.], 2019. v.2, p.115–120.

Fabrice Bellard. **QuickJS Javascript Engine**. Accessed: Mar. 2023, https://bellard.org/quickjs.

FARD, A. M.; MESBAH, A. JSNOSE: Detecting JavaScript Code Smells. In: IEEE 13TH INTERNATIONAL WORKING CONFERENCE ON SOURCE CODE ANALYSIS AND MANIPULATION (SCAM), 2013., 2013. **Anais...** [S.l.: s.n.], 2013. p.116–125.

FARD, A. M.; MESBAH, A. JavaScript: The (un) covered parts. In: IEEE INTERNATIONAL CONFERENCE ON SOFTWARE TESTING, VERIFICATION AND VALIDATION (ICST), 2017., 2017. **Anais...** [S.l.: s.n.], 2017. p.230–240.

FARHAN, L. et al. A survey on the challenges and opportunities of the Internet of Things (IoT). In: ELEVENTH INTERNATIONAL CONFERENCE ON SENSING TECHNOLOGY (ICST), 2017., 2017. **Anais...** [S.l.: s.n.], 2017. p.1–5.

FARSHIDI, S.; JANSEN, S.; DELDAR, M. A decision model for programming language ecosystem selection: Seven industry case studies. **Information and Software Technology**, [S.l.], v.139, p.106640, 2021.

FLANAGAN, D. **JavaScript - The Definitive Guide**. 7.ed. Sebastopol, CA: O'Reilly Media, 2020.

FLANAGAN, D. **JavaScript**: the definitive guide. Seventh Edition.ed. [S.l.]: " O'Reilly Media, Inc.", 2020.

GASCON-SAMSON, J.; JUNG, K.; PATTABIRAMAN, K. Poster: Towards a Distributed and Self-Adaptable Cloud-Edge Middleware. In: IEEE/ACM SYMPOSIUM ON EDGE COMPUTING (SEC), 2018., 2018. **Anais...** [S.l.: s.n.], 2018. p.338–340.

GASCON-SAMSON, J. et al. Thingsmigrate: Platform-independent migration of stateful javascript iot applications. In: EUROPEAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING (ECOOP 2018), 32., 2018. **Anais...** [S.l.: s.n.], 2018.

GASCON-SAMSON, J.; RAFIUZZAMAN, M.; PATTABIRAMAN, K. Thingsjs: Towards a flexible and self-adaptable middleware for dynamic and heterogeneous iot environments. In: WORKSHOP ON MIDDLEWARE AND APPLICATIONS FOR THE INTERNET OF THINGS, 4., 2017. **Proceedings...** [S.l.: s.n.], 2017. p.11–16.

GASCON-SAMSON, J.; RAFIUZZAMAN, M.; PATTABIRAMAN, K. SmartJS: dynamic and self-adaptable runtime middleware for next-generation IoT systems. In: COMPANION OF THE 2017 ACM SIGPLAN INTERNATIONAL CONFERENCE ON SYSTEMS, PROGRAMMING, LANGUAGES, AND APPLICATIONS: SOFTWARE FOR HUMANITY, 2017. **Proceedings...** [S.l.: s.n.], 2017. p.51–52.

GAVRIN, E.; LEE, S.-J.; AYRAPETYAN, R.; SHITOV, A. Ultra lightweight JavaScript engine for internet of things. In: COMPANION PROCEEDINGS OF THE 2015 ACM SIGPLAN INTERNATIONAL CONFERENCE ON SYSTEMS, PROGRAMMING, LANGUAGES AND APPLICATIONS: SOFTWARE FOR HUMANITY, 2015. **Anais...** [S.l.: s.n.], 2015. p.19–20.

GERHORST, L.; REIF, S.; HERZOG, B.; HöNIG, T. EnergyBudgets: Integrating Physical Energy Measurement Devices into Systems Software. In: X BRAZILIAN SYMPOSIUM ON COMPUTING SYSTEMS ENGINEERING (SBESC), 2020., 2020. **Anais...** [S.l.: s.n.], 2020. p.1–8.

GHOSH, D.; JIN, F.; MAHESWARAN, M. JADE: A unified programming framework for things, web, and cloud. In: INTERNATIONAL CONFERENCE ON THE INTERNET OF THINGS (IOT), 2014., 2014. **Anais...** [S.l.: s.n.], 2014. p.73–78.

GONZÁLEZ GARCÍA, C.; MEANA LLORIÁN, D.; PELAYO GARCÍA-BUSTELO, B. C.; CUEVA LOVELLE, J. M. A review about smart objects, sensors, and actuators. **International Journal of Interactive Multimedia and Artificial Intelligence**, [S.l.], 2017.

GONZáLEZ GARCíA, C.; ZHAO, L.; GARCíA-DíAZ, V. A User-Oriented Language for Specifying Interconnections Between Heterogeneous Objects in the Internet of Things. **IEEE Internet of Things Journal**, [S.l.], v.6, n.2, p.3806–3819, 2019.

Google. **V8 JavaScript engine**. Accessed: Mar. 2023, `https://v8.dev/`.

Google Actions. **Google Smart Home Platform**. Accessed: Mar. 2023, `https://developers.google.com/assistant/smarthome`.

GOSLING, J.; JOY, B.; STEELE, G.; BRACHA, G. **The Java language specification**. [S.l.]: Addison-Wesley Professional, 2000.

GRESSL, L.; STEGER, C.; NEFFE, U. Security Driven Design Space Exploration for Embedded Systems. In: FORUM FOR SPECIFICATION AND DESIGN LANGUAGES (FDL), 2019., 2019. **Anais...** [S.l.: s.n.], 2019. p.1–8.

GRUNERT, K. Overview of JavaScript Engines for Resource-Constrained Microcontrollers. In: INTERNATIONAL CONFERENCE ON SMART AND SUSTAINABLE TECHNOLOGIES (SPLITECH), 2020., 2020. **Anais...** [S.l.: s.n.], 2020. p.1–7.

GUBBI, J.; BUYYA, R.; MARUSIC, S.; PALANISWAMI, M. Internet of Things (IoT): A vision, architectural elements, and future directions. **Future generation computer systems**, [S.l.], v.29, n.7, p.1645–1660, 2013.

GUINARD, D.; TRIFA, V.; WILDE, E. et al. A resource oriented architecture for the Web of Things. In: IOT, 2010. **Anais...** [S.l.: s.n.], 2010. p.1–8.

HALSTEAD, M. H. **Elements of Software Science (Operating and Programming Systems Series)**. USA: Elsevier Science Inc., 1977.

HAN, Z.; DEVARAJEGOWDA, K.; WERNER, M.; ECKER, W. Towards a Python-Based One Language Ecosystem for Embedded Systems Automation. In: IEEE NORDIC CIRCUITS AND SYSTEMS CONFERENCE (NORCAS): NORCHIP AND INTERNATIONAL SYMPOSIUM OF SYSTEM-ON-CHIP (SOC), 2019., 2019. **Anais...** [S.l.: s.n.], 2019. p.1–7.

HEATH, S. **Embedded systems design**. [S.l.]: Elsevier, 2002.

HENNESSY, J. L.; PATTERSON, D. A. **Computer Architecture, Sixth Edition**: A Quantitative Approach. 6th.ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2017.

HEO, Y. J.; OH, S. M.; CHIN, W. S.; JANG, J. W. A lightweight platform implementation for Internet of Things. In: INTERNATIONAL CONFERENCE ON FUTURE INTERNET OF THINGS AND CLOUD, 2015., 2015. **Anais...** [S.l.: s.n.], 2015. p.526–531.

HIRASAWA, Y.; IWASAKI, H.; UGAWA, T.; ONOZAWA, H. Generating Virtual Machine Code of JavaScript Engine for Embedded Systems. **Journal of Information Processing**, [S.l.], v.30, n.0, p.679–693, 2022.

HODDIE, P.; PRADER, L. **IoT Development for ESP32 and ESP8266 with JavaScript**: A Practical Guide to XS and the Moddable SDK. [S.l.: s.n.], 2020.

HONG, G.; SHIN, D. Segment-Based Multiple-Base Compressed Addressing for Flexible JavaScript Heap Allocation. **IEEE Access**, [S.l.], v.8, p.185405–185415, 2020.

IoT Analytics. **Global IoT market size to grow 19% in 2023—IoT shows resilience despite economic downturn**. Accessed: Mar. 2023, https://iot-analytics.com/iot-market-size.

IWATA, K.; NAKASHIMA, T.; ANAN, Y.; ISHII, N. Applying Machine Learning Classification to Determining Outliers in Effort for Embedded Software Development Projects. In: INTERNATIONAL CONFERENCE ON COMPUTATIONAL SCIENCE/INTELLIGENCE AND APPLIED INFORMATICS (CSII), 2019., 2019. **Anais...** [S.l.: s.n.], 2019. p.78–83.

JAIMINI, U.; DHANIWALA, M. JavaScript empowered Internet of things. In: INTERNATIONAL CONFERENCE ON COMPUTING FOR SUSTAINABLE GLOBAL DEVELOPMENT (INDIACOM), 2016., 2016. **Anais...** [S.l.: s.n.], 2016. p.2373–2377.

JerryScript. **JavaScript Engine for Internet of Things**. Accessed: Mar. 2023, https://jerryscript.net.

JOSHI, P. V.; GURUMURTHY, K. Analysing and improving the performance of software code for Real Time Embedded systems. In: INTERNATIONAL CONFERENCE ON DEVICES, CIRCUITS AND SYSTEMS (ICDCS), 2014., 2014. **Anais...** [S.l.: s.n.], 2014. p.1–5.

JUNG, K. et al. ThingsMigrate: Platform-independent migration of stateful JavaScript Internet of Things applications. **Software: Practice and Experience**, [S.l.], v.51, n.1, p.117–155, 2021.

Khan, Faiz and Foley-Bourgon, Vincent and Kathrotia, Sujay and Lavoie, Erick. **Ostrich Benchmark Suite**. Disponível em: <https://github.com/Sable/Ostrich>.

KIENLE, H. M.; KRAFT, J.; NOLTE, T. System-specific static code analyses: a case study in the complex embedded systems domain. **Software quality journal**, [S.l.], v.20, n.2, p.337–367, 2012.

KIM, M.; JEONG, H.-J.; MOON, S.-M. **Small Footprint JavaScript Engine**. [S.l.: s.n.], 2017. p.103–116.

KIRCHHOF, J. C.; RUMPE, B.; SCHMALZING, D.; WORTMANN, A. MontiThings: Model-Driven Development and Deployment of Reliable IoT Applications. **Journal of Systems and Software**, [S.l.], v.183, p.111087, 2022.

KITCHENHAM, B. et al. Systematic literature reviews in software engineering–a tertiary study. **Information and software technology**, [S.l.], v.52, n.8, p.792–805, 2010.

KNAPPMEYER, M. et al. Survey of context provisioning middleware. **IEEE Communications Surveys & Tutorials**, [S.l.], v.15, n.3, p.1492–1519, 2013.

KOLIOS, P.; PANAYIOTOU, C.; ELLINAS, G.; POLYCARPOU, M. Data-Driven Event Triggering for IoT Applications. **IEEE Internet of Things Journal**, [S.l.], v.3, n.6, p.1146–1158, 2016.

KOOPMAN, P.; LUBBERS, M.; PLASMEIJER, R. A Task-Based DSL for Microcomputers. In: REAL WORLD DOMAIN SPECIFIC LANGUAGES WORKSHOP 2018, 2018, New York, NY, USA. **Proceedings...** Association for Computing Machinery, 2018. (RWDSL2018).

KRAELING, M. Embedded software programming and implementation guidelines. In: **Software Engineering for Embedded Systems**. [S.l.]: Elsevier, 2013. p.183–204.

KRISHNAMURTHY, J.; MAHESWARAN, M. Chapter 5 - Programming frameworks for Internet of Things. In: BUYYA, R.; Vahid Dastjerdi, A. (Ed.). **Internet of Things**. [S.l.]: Morgan Kaufmann, 2016. p.79–102.

KWON, J.-w.; MOON, S.-M. Work-in-progress: JSDelta: serializing modified javascript states for state sharing. In: INTERNATIONAL CONFERENCE ON EMBEDDED SOFTWARE (EMSOFT), 2017., 2017. **Anais...** [S.l.: s.n.], 2017. p.1–2.

L, B.; JULIAN, A. Design and implementation of Automated Blood Bank using embedded systems. In: INTERNATIONAL CONFERENCE ON INNOVATIONS IN INFORMATION, EMBEDDED AND COMMUNICATION SYSTEMS (ICIIECS), 2015., 2015. **Anais...** [S.l.: s.n.], 2015. p.1–6.

LACERDA, G.; PETRILLO, F.; PIMENTA, M.; GUéHéNEUC, Y. G. Code smells and refactoring: A tertiary systematic review of challenges and observations. **Journal of Systems and Software**, [S.l.], v.167, p.110610, 2020.

LEE, E. A.; SESHIA, S. A. **Introduction to embedded systems**: A cyber-physical systems approach. [S.l.]: Mit Press, 2017.

LEE, H. et al. Open software platform for companion IoT devices. In: IEEE INTERNATIONAL CONFERENCE ON CONSUMER ELECTRONICS (ICCE), 2017., 2017. **Anais...** [S.l.: s.n.], 2017. p.394–395.

LI, D.; HUANG, B.; CUI, L.; XU, Z. WebletScript: A Lightweight Distributed JavaScript Engine for Internet of Things. In: IEEE GLOBAL COMMUNICATIONS CONFERENCE (GLOBECOM), 2018., 2018. **Anais...** [S.l.: s.n.], 2018. p.1–6.

LIU, Y. JSOptimizer: An Extensible Framework for JavaScript Program Optimization. In: IEEE/ACM 41ST INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING: COMPANION PROCEEDINGS (ICSE-COMPANION), 2019., 2019. **Anais...** [S.l.: s.n.], 2019. p.168–170.

LUBBERS, M.; KOOPMAN, P.; PLASMEIJER, R. **Writing Internet of Things Applicatations with Task-Oriented Programming**. [S.l.]: arXiv, 2022. Disponível em: <https://arxiv.org/abs/2212.04193>.

LóKI., G.; GáL., P. **JavaScript Guidelines for JavaScript Programmers - A Comprehensive Guide for Performance Critical JS Programs**. [S.l.]: SciTePress, 2018. 397-404p.

MARWEDEL, P. **Embedded system design**: embedded systems foundations of cyber-physical systems, and the internet of things. [S.l.]: Springer Nature, 2021.

MCCABE, T. A Complexity Measure. **IEEE Transactions on Software Engineering**, [S.l.], v.SE-2, n.4, p.308–320, 1976.

MIORANDI, D.; SICARI, S.; DE PELLEGRINI, F.; CHLAMTAC, I. Internet of things: Vision, applications and research challenges. **Ad hoc networks**, [S.l.], v.10, n.7, p.1497–1516, 2012.

MIRAZ, M. H.; ALI, M.; EXCELL, P. S.; PICKING, R. A review on Internet of Things (IoT), Internet of everything (IoE) and Internet of nano things (IoNT). In: INTERNET TECHNOLOGIES AND APPLICATIONS (ITA), 2015., 2015. **Anais...** [S.l.: s.n.], 2015. p.219–224.

Moddable Github. **Moddable-OpenSource**. Accessed: Mar. 2023, `https://github.com/Moddable-OpenSource/moddable`.

Moddable Tech. **Moddable Tech**: Modern software development for microcontrollers. Accessed: Mar. 2023, `https://www.moddable.com/`.

Mongoose OS. **mJS**: Restricted JavaScript Engine. Accessed: Mar. 2023, `https://github.com/cesanta/mjs`.

MORALES, R.; SABORIDO, R.; GUÉHÉNEUC, Y.-G. Momit: Porting a javascript interpreter on a quarter coin. **IEEE Transactions on Software Engineering**, [S.l.], 2020.

MORALES, R.; SABORIDO, R.; GUéHéNEUC, Y.-G. MoMIT: Porting a JavaScript Interpreter on a Quarter Coin. **IEEE Transactions on Software Engineering**, [S.l.], v.47, n.12, p.2771–2785, 2021.

Mozilla Developer Network. **Documenting web technologies**: JavaScript. Accessed: Mar. 2023, `https://developer.mozilla.org/en-US/docs/Web/JavaScript`.

Mozilla Foundation. **Rhino an Implementation of JavaScript in Java**. Accessed: Mar. 2023, `https://github.com/mozilla/rhino`.

Mozilla Foundation. **SpiderMonkey**: JavaScript and WebAssembly Engine. Accessed: Mar. 2023, `https://developer.apple.com/documentation/javascriptcore`.

MUDALIAR, M. D.; SIVAKUMAR, N. IoT based real time energy monitoring system using Raspberry Pi. **Internet of Things**, [S.l.], v.12, p.100292, 2020.

NAHAS, M.; NAHHAS, A. M. Ways for Implementing Highly-Predictable Embedded Systems Using Time-Triggered Co-Operative (TTC) Architectures. In: TANAKA, K. (Ed.). **Embedded Systems**. Rijeka: IntechOpen, 2012.

NAKAGAWA, H. et al. Embedded System Evolution in IoT System Development Based on MAPE-K Loop Mechanism. **arXiv preprint arXiv:2205.13375**, [S.l.], 2022.

NAMIOT, D.; SNEPS-SNEPPE, M. On iot programming. **International Journal of Open Information Technologies**, [S.l.], v.2, n.10, p.25–28, 2014.

NASCIMENTO, R.; BRITO, A.; HORA, A.; FIGUEIREDO, E. JavaScript API Deprecation in the Wild: A First Assessment. In: IEEE 27TH INTERNATIONAL CONFERENCE ON SOFTWARE ANALYSIS, EVOLUTION AND REENGINEERING (SANER), 2020., 2020. **Anais...** [S.l.: s.n.], 2020. p.567–571.

Node.js. **An Asynchronous Event-driven JavaScript Runtime**. Accessed: Mar. 2023, `https://nodejs.org/en/docs`.

Nordic Semiconductor. **Power Profiler Kit II - Documentation**. Accessed: Mar. 2023, `https://infocenter.nordicsemi.com`.

OBERMAISSER, R. **Event-Triggered and Time-Triggered Control Paradigms**. [S.l.]: Springer US, 2005.

OLIPHANT, T. E. Python for Scientific Computing. **Computing in Science Engineering**, [S.l.], v.9, n.3, p.10–20, 2007.

OLIVEIRA, F. Student Research Abstract: A Hybrid Approach to Design Embedded Software Using JavaScript's Non-Blocking Principle. In: ACM/SIGAPP SYMPOSIUM ON APPLIED COMPUTING, 38., 2023, New York, NY, USA. **Proceedings...** Association for Computing Machinery, 2023. p.732–735. (SAC '23).

OLIVEIRA, F. L.; MATTOS, J. C. B. JSGuide: A Tool to Improve JavaScript Algorithms Focusing on IoT Devices. In: SYMPOSIUM ON INTERNET OF THINGS (SIOT), 2022., 2022. **Anais...** [S.l.: s.n.], 2022. p.1–4.

OLIVEIRA, F. L.; MATTOS, J. C. B. JSEVAsync: An Asynchronous Event-based Framework to Energy Saving on IoT Devices. In: XII BRAZILIAN SYMPOSIUM ON COMPUTING SYSTEMS ENGINEERING (SBESC), 2022., 2022. **Anais...** [S.l.: s.n.], 2022. p.1–7.

OLIVEIRA, F. L.; PARIZI, R. R.; MATTOS, J. C. B. de. Improving Developer Productivity on Internet of Things using JavaScript. In: INTERNATIONAL CONFERENCE ON INTERNET OF THINGS, BIG DATA AND SECURITY, IOTBDS 2022, ONLINE STREAMING, APRIL 22-24, 2022, 7., 2022. **Proceedings...** SCITEPRESS, 2022. p.223–230.

OLIVEIRA, F.; MATTOS, J. State-of-the-Art Javascript Language for Internet of Things. In: ESTENDIDOS DO IX SIMPóSIO BRASILEIRO DE ENGENHARIA DE SISTEMAS COMPUTACIONAIS, 2019, Porto Alegre, RS, Brasil. **Anais...** SBC, 2019. p.149–154.

OLIVEIRA, F.; MATTOS, J. Analysis of WebAssembly as a Strategy to Improve JavaScript Performance on IoT Environments. In: ESTENDIDOS DO X SIMPÓSIO BRASILEIRO DE ENGENHARIA DE SISTEMAS COMPUTACIONAIS, 2020. **Anais...** [S.l.: s.n.], 2020. p.133–138.

OLIVEIRA, F.; MATTOS, J. Webassemby: Uma Estratégia Para Melhorar O Desempenho Da Aplicação JavaScript Em Ambientes IoT. In: IN ANAIS 6 SIIEPE – SEMANA INTEGRADA UFPEL / XXII ENPOS – ENCONTRO DE PóS-GRADUAçãO., 2020, Pelotas, RS, Brasil. **Anais...** SBC, 2020.

OLIVEIRA, F.; MATTOS, J. Analysis of WebAssembly as a Strategy to Improve JavaScript Performance on IoT Environments. In: ESTENDIDOS DO X SIMPóSIO BRASILEIRO DE ENGENHARIA DE SISTEMAS COMPUTACIONAIS, 2020, Porto Alegre, RS, Brasil. **Anais...** SBC, 2020. p.133–138.

OLIVEIRA, L.; MATTOS, J. C. B.; BRISOLARA, L. Survey of Memory Optimization Techniques for Embedded Systems. In: III BRAZILIAN SYMPOSIUM ON COMPUTING SYSTEMS ENGINEERING, 2013., 2013. **Anais...** [S.l.: s.n.], 2013. p.65–70.

OLIVEIRA, M. F. et al. Software quality metrics and their impact on embedded software. In: TH INTERNATIONAL WORKSHOP ON MODEL-BASED METHODOLOGIES FOR PERVASIVE AND EMBEDDED SOFTWARE, 2008., 2008. **Anais...** [S.l.: s.n.], 2008. p.68–77.

OLIVEIRA, M. F. et al. Software quality metrics and their impact on embedded software. In: TH INTERNATIONAL WORKSHOP ON MODEL-BASED METHODOLOGIES FOR PERVASIVE AND EMBEDDED SOFTWARE, 2008., 2008. **Anais...** [S.l.: s.n.], 2008. p.68–77.

PAPADOPOULOS, L. et al. Interrelations between software quality metrics, performance and energy consumption in embedded applications. In: INTERNATIONAL WORKSHOP ON SOFTWARE AND COMPILERS FOR EMBEDDED SYSTEMS, 21., 2018. **Proceedings...** [S.l.: s.n.], 2018. p.62–65.

PARK, H.; JUNG, W.; MOON, S.-M. JavaScript ahead-of-time compilation for embedded web platform. In: IEEE SYMPOSIUM ON EMBEDDED SYSTEMS FOR REAL-TIME MULTIMEDIA (ESTIMEDIA), 2015., 2015. **Anais...** [S.l.: s.n.], 2015. p.1–9.

PATTANAYAK, B. K.; PATRA, S. K.; PUTHAL, B. Optimizing AST Node for Java Script Compiler A lightweight Interpreter for Embedded Device. **Journal of Computers**, [S.l.], v.8, n.2, Feb. 2013.

PETERSEN, K.; FELDT, R.; MUJTABA, S.; MATTSSON, M. Systematic mapping studies in software engineering. In: EASE, 2008. **Anais...** [S.l.: s.n.], 2008. v.8, p.68–77.

PETERSON, B.; VOGEL, B. Prototyping the Internet of Things with Web Technologies: Is It Easy? In: IEEE INTERNATIONAL CONFERENCE ON PERVASIVE COMPUTING AND COMMUNICATIONS WORKSHOPS (PERCOM WORKSHOPS), 2018., 2018. **Anais...** [S.l.: s.n.], 2018. p.518–522.

PINHO, A.; COUTO, L.; OLIVEIRA, J. Towards Rust for Critical Systems. In: IEEE INTERNATIONAL SYMPOSIUM ON SOFTWARE RELIABILITY ENGINEERING WORKSHOPS (ISSREW), 2019., 2019. **Anais...** [S.l.: s.n.], 2019. p.19–24.

PRABHU, S. S.; KAPIL, H.; LAKSHMAIAH, S. H. Safety Critical Embedded Software: Significance and Approach to Reliability. In: INTERNATIONAL CONFERENCE ON ADVANCES IN COMPUTING, COMMUNICATIONS AND INFORMATICS (ICACCI), 2018., 2018. **Anais...** [S.l.: s.n.], 2018. p.449–455.

PULIAFITO, C. et al. Fog computing for the internet of things: A Survey. **ACM Transactions on Internet Technology (TOIT)**, [S.l.], v.19, n.2, p.18, 2019.

RAHMAN, L. F.; OZCELEBI, T.; LUKKIEN, J. Understanding IoT Systems: A Life Cycle Approach. **Procedia Computer Science**, [S.l.], v.130, p.1057–1062, 2018. The 9th International Conference on Ambient Systems, Networks and Technologies (ANT 2018) / The 8th International Conference on Sustainable Energy Information Technology (SEIT-2018) / Affiliated Workshops.

Raspberry Pi computers and microcontrollers. **Raspberry Pi Documentation**. Accessed: Mar. 2023, `https://www.raspberrypi.com/documentation`.

RC-switch library. **RC-switch**. Accessed: Mar. 2023, `https://github.com/sui77/rc-switch`.

RFC 7228: Terminology for constrained-node network. **Internet Engineering Task Force (IETF)**. Accessed: Mar. 2023, `https://tools.ietf.org/html/rfc7228`.

RILISKIS, L.; HONG, J.; LEVIS, P. Ravel: Programming iot applications as distributed models, views, and controllers. In: INTERNATIONAL WORKSHOP ON INTERNET OF THINGS TOWARDS APPLICATIONS, 2015., 2015. **Proceedings...** [S.l.: s.n.], 2015. p.1–6.

SABOURY, A.; MUSAVI, P.; KHOMH, F.; ANTONIOL, G. An empirical study of code smells in JavaScript projects. In: IEEE 24TH INTERNATIONAL CONFERENCE ON SOFTWARE ANALYSIS, EVOLUTION AND REENGINEERING (SANER), 2017., 2017. **Anais...** [S.l.: s.n.], 2017. p.294–305.

SAHOO, S. S. et al. Emergent Design Challenges for Embedded Systems and Paths Forward: Mixed-Criticality, Energy, Reliability and Security Perspectives. In: INTERNATIONAL CONFERENCE ON HARDWARE/SOFTWARE CODESIGN AND SYSTEM SYNTHESIS, 2021., 2021, New York, NY, USA. **Proceedings...** Association for Computing Machinery, 2021. p.1–10. (CODES/ISSS '21).

SAHU, A.; SINGH, A. Securing IoT devices using JavaScript based sandbox. In: IEEE INTERNATIONAL CONFERENCE ON RECENT TRENDS IN ELECTRONICS, INFORMATION & COMMUNICATION TECHNOLOGY (RTEICT), 2016., 2016. **Anais...** [S.l.: s.n.], 2016. p.1476–1482.

SALIHBEGOVIC, A.; ETEROVIC, T.; KALJIC, E.; RIBIC, S. Design of a domain specific language and IDE for Internet of things applications. In: INTERNATIONAL CONVENTION ON INFORMATION AND COMMUNICATION TECHNOLOGY, ELECTRONICS AND MICROELECTRONICS (MIPRO), 2015., 2015. **Anais...** [S.l.: s.n.], 2015. p.996–1001.

SALMAN, A. J.; AL-JAWAD, M.; TAMEEMI, W. A. Domain-Specific Languages for IoT: Challenges and Opportunities. **IOP Conference Series: Materials Science and Engineering**, [S.l.], v.1067, n.1, p.012133, feb 2021.

SANGIOVANNI-VINCENTELLI, A.; MARTIN, G. Platform-based design and software design methodology for embedded systems. **IEEE Design & Test of Computers**, [S.l.], v.18, n.6, p.23–33, 2001.

SCHELER, F.; SCHROEDER-PREIKSCHAT, W. Time-Triggered vs. Event-Triggered: A matter of configuration? In: ITG FA 6.2 WORKSHOP ON MODEL-BASED TESTING, GI/ITG WORKSHOP ON NON-FUNCTIONAL PROPERTIES OF EMBEDDED SYSTEMS, 13TH GI/ITG CONFERENCE MEASURING, MODELLING, AND EVALUATION OF COMPUTER AND COMMUNICATIONS, 2006. **Anais...** [S.l.: s.n.], 2006. p.1–6.

SCHILDT, H. **C completo e total**. [S.l.]: Makron, 1997.

SCOTT, M. L. **Programming language pragmatics**. [S.l.]: Morgan Kaufmann, 2000.

SELAKOVIC, M.; PRADEL, M. Performance Issues and Optimizations in JavaScript: An Empirical Study. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 38., 2016, New York, NY, USA. **Proceedings...** Association for Computing Machinery, 2016. p.61–72. (ICSE '16).

Severin, T.; Culic, I.; Radovici, A. Enabling High-Level Programming Languages on IoT Devices. In: ROEDUNET CONFERENCE: NETWORKING IN EDUCATION AND RESEARCH (ROEDUNET), 2020., 2020. **Anais...** [S.l.: s.n.], 2020. p.1–6.

SHULL, T.; CHOI, J.; GARZARAN, M. J.; TORRELLAS, J. NoMap: Speeding-Up JavaScript Using Hardware Transactional Memory. In: IEEE INTERNATIONAL SYMPOSIUM ON HIGH PERFORMANCE COMPUTER ARCHITECTURE (HPCA), 2019., 2019. **Anais...** [S.l.: s.n.], 2019. p.412–425.

Software Engineering Research Group at Politecnico di Torino. **Halstead Metrics Tool**. Accessed: Mar. 2023, `https://github.com/SoftengPoliTo/Halstead-Metrics-Tool`.

Stack Overflow. **Stack Overflow Annual Developer Survey 2022**. Accessed: Mar. 2023, `https://survey.stackoverflow.co/2022`.

STEBBINS, R. A. **Exploratory research in the social sciences**. [S.l.]: Sage, 2001. v.48.

TAIVALSAARI, A.; MIKKONEN, T. A roadmap to the programmable world: software challenges in the IoT era. **IEEE Software**, [S.l.], v.34, n.1, p.72–80, 2017.

Taivalsaari, A.; Mikkonen, T. A Taxonomy of IoT Client Architectures. **IEEE Software**, [S.l.], v.35, n.3, p.83–88, May 2018.

Terry Yin. **Lizard Cyclomatic Complexity Analyzer**. Accessed: Mar. 2023, `https://github.com/terryyin/lizard`.

THAKER, N.; BABU, S. S. K. Analysis of event-triggered and time-triggered architecture for a reliable embedded system. In: INTERNATIONAL CONFERENCE ON COMPUTING, COMMUNICATION AND NETWORKING TECHNOLOGIES (ICCCNT), 2017., 2017. **Anais...** [S.l.: s.n.], 2017. p.1–5.

THOLE, S. P.; RAMU, P. Design space exploration and optimization using self-organizing maps. **Structural and Multidisciplinary Optimization**, [S.l.], v.62, n.3, p.1071–1088, 2020.

TORRES, D.; DIAS, J. P.; RESTIVO, A.; FERREIRA, H. S. Real-time Feedback in Node-RED for IoT Development: An Empirical Study. In: IEEE/ACM 24TH INTERNATIONAL SYMPOSIUM ON DISTRIBUTED SIMULATION AND REAL TIME APPLICATIONS (DS-RT), 2020., 2020. **Anais...** [S.l.: s.n.], 2020. p.1–8.

Transforma Insights Institute. **The internet of things (IoT) market 2019-2030**. Accessed: Mar. 2023, https://transformainsights.com/news/iot-market-24-billion-usd15-trillion-revenue-2030.

UGAWA, T.; IWASAKI, H.; KATAOKA, T. eJSTK: Building JavaScript virtual machines with customized datatypes for embedded systems. **Journal of Computer Languages**, [S.l.], v.51, p.261–279, 2019.

UGAWA, T.; MARR, S.; JONES, R. Profile Guided Offline Optimization of Hidden Class Graphs for JavaScript VMs in Embedded Systems. In: ACM SIGPLAN INTERNATIONAL WORKSHOP ON VIRTUAL MACHINES AND INTERMEDIATE LANGUAGES, 14., 2022, New York, NY, USA. **Proceedings...** Association for Computing Machinery, 2022. p.25–35. (VMIL 2022).

VALSAMAKIS, Y.; SAVIDIS, A. Personal applications in the internet of things through visual end-user programming. **Digital Marketplaces Unleashed**, [S.l.], p.809–821, 2018.

VANOMMESLAEGHE, Y.; DENIL, J.; VAN ACKER, B.; DE MEULENAERE, P. Automatic Generation of Workflows for Efficient Design Space Exploration for Cyber-Physical Systems. In: IEEE INTERNATIONAL CONFERENCES ON INTERNET OF THINGS (ITHINGS) AND IEEE GREEN COMPUTING COMMUNICATIONS (GREENCOM) AND IEEE CYBER, PHYSICAL SOCIAL COMPUTING (CPSCOM) AND IEEE SMART DATA (SMARTDATA) AND IEEE CONGRESS ON CYBERMATICS (CYBERMATICS), 2021., 2021. **Anais...** [S.l.: s.n.], 2021. p.346–351.

VERMA, M.; MARWEDEL, P. **Advanced Memory Optimization Techniques for Low-Power Embedded Processors**. [S.l.]: Springer Netherlands, 2007.

WANG, B.-Y.; YEN, Y.-C.; CHENG, Y. C. Specifying Internet of Things Behaviors in Behavior-Driven Development: Concurrency Enhancement and Tool Support. **Applied Sciences**, [S.l.], v.13, n.2, 2023.

WANG, M. B.; MANESH, D.; HU, R.; LEE, S. W. iThem: Programming Internet of Things Beyond Trigger-Action Pattern. In: THE ADJUNCT PUBLICATION OF THE 35TH ANNUAL ACM SYMPOSIUM ON USER INTERFACE SOFTWARE AND TECHNOLOGY, 2022. **Anais...** ACM, 2022.

WANG, R. et al. Efficient Asynchronous Communication between Virtual Machines in Embedded Systems. In: IEEE 19TH INTERNATIONAL CONFERENCE ON HIGH PERFORMANCE COMPUTING AND COMMUNICATIONS; IEEE 15TH INTERNATIONAL CONFERENCE ON SMART CITY; IEEE 3RD INTERNATIONAL CONFERENCE ON DATA SCIENCE AND SYSTEMS (HPCC/SMARTCITY/DSS), 2017., 2017. **Anais...** [S.l.: s.n.], 2017. p.603–604.

WOLF, M. Chapter 1 - Embedded Computing. In: WOLF, M. (Ed.). **Computers as Components (Fourth Edition)**. Fourth Edition.ed. [S.l.]: Morgan Kaufmann, 2017. p.1–54. (The Morgan Kaufmann Series in Computer Architecture and Desi).

WOLF, M. Computing in the real world is the grandest of challenges. **Computer**, [S.l.], v.51, n.5, p.90–91, 2018.

WOLF, W.; KANDEMIR, M. Memory system optimization of embedded software. **Proceedings of the IEEE**, [S.l.], v.91, n.1, p.165–182, 2003.

World Wide Web Consortium. **JavaScript Web APIs**. Accessed: Mar. 2023, `https://www.w3.org/standards/webdesign/script`.

YAMASHITA, A. How Good Are Code Smells for Evaluating Software Maintainability? Results from a Comparative Case Study. In: IEEE INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE, 2013., 2013. **Anais...** [S.l.: s.n.], 2013. p.566–571.

APPENDIX A: MICROBENCHMARKS

```
export class Rest {
  static standard() {
    const numbers = [142, 12, 56, 34, -123];
    const v1 = numbers[0];
    const v2 = numbers[1];
    const rest = numbers.slice(2);
    return rest;
  }

  static modified() {
    const numbers = [142, 12, 56, 34, -123];
    const { v1, v2, ...rest } = numbers;
    return rest;
  }
}
```

Listing 7.1 – Rest benchmark

```
export class Spread {
  static standard() {
    const document = {
      id: 1,
      title: "The document title",
    };
    const newDocument = Object.assign(document, { attached: "
    file.pdf" });
    return newDocument;
  }

  static modified() {
    const document = {
      id: 1,
      title: "The document title",
    };
    const newDocument = {
      ...document,
      attached: "file.pdf",
    };
    return newDocument;
  }
```

```
}
```

Listing 7.2 – Spread benchmark

```
export class FromEntries {
  static standard() {
    const data = [
      ["age", 18],
      ["name", "Philip"],
    ];
    const person = {};
    person[data[0][0]] = data[0][1];
    person[data[1][0]] = data[1][1];
    return person;
  }

  static modified() {
    const data = [
      ["age", 18],
      ["name", "Philip"],
    ];
    return Object.fromEntries(data);
  }
}
```

Listing 7.3 – FromEntries benchmark

```
export class Flat {
  static standard() {
    const arr = [1, 22, 13, ["abc", () => {}], [-12, 2.3, true
    ]];
    return arr.reduce((acc, val) => acc.concat(val), []);
  }

  static modified() {
    const arr = [1, 22, 13, ["abc", () => {}], [-12, 2.3, true
    ]];
    return arr.flat(1);
  }
}
```

Listing 7.4 – Flat benchmark

```
export class StringTrim {
  static standard() {
```

```
    let aux = "      hello world";
    return aux.replace(/^\s+/g, "");
  }

  static modified() {
    let aux = "      hello world";
    return aux.trimStart();
  }
}
```

Listing 7.5 – TrimStart / trimEnd benchmark

```
export class MatchAll {
  static standard() {
    const msg = "Split the sentence by white space.";
    const regexp = new RegExp(/[^\s]+/g);
    let value;
    while ((value = regexp.exec(msg)) !== null) {
      console.log(value[0]);
    }
    return null;
  }

  static modified() {
    const msg = "Split the sentence by white space.";
    for (const value of msg.matchAll(/[^\s]+/g)) {
      console.log(value[0]);
    }
    return null;
  }
}
```

Listing 7.6 – MatchAll benchmark

```
export class Nullish {
  static standard() {
    const points = 0;
    // needs to return zero
    return points ? points : -1;
  }

  static modified() {
    const points = 0;
    // needs to return zero
```

```
    return points ?? -1;
  }
}
```

Listing 7.7 – Nullish coalescing benchmark

```
export class OptionalChaining {
  static standard() {
    const entity = {
      person: {
        name: "Philip",
        address: {
          street: "5th Avenue",
        },
      },
    };
    if (entity.person && entity.person.address) {
      console.log(entity.person.address.street);
    }
    return null;
  }

  static modified() {
    const entity = {
      person: {
        name: "Philip",
        address: {
          street: "5th Avenue",
        },
      },
    };
    console.log(entity.person?.address?.street);
    return null;
  }
}
```

Listing 7.8 – Optional chaining benchmark

```
export class ReplaceAll {
  static standard() {
    const aux = "This is the JS language. Run JS anywhere.";
    let replaced = aux.replace(/JS/g, "JavaScript");
    return replaced;
  }
```

```
7
8   static modified() {
9     const aux = "This is the JS language. Run JS anywhere.";
10    let replaced = aux.replaceAll("JS", "JavaScript");
11    return replaced;
12  }
13 }
```

Listing 7.9 – ReplaceAll benchmark

```
1 export class LogicalAssignment {
2   static standard() {
3     let value = 10;
4     if (value) {
5       value = 20;
6     }
7     return value;
8   }
9
10  static modified() {
11    let value = 10;
12    value &&= 20;
13    return value;
14  }
15 }
```

Listing 7.10 – Logical assignment benchmark

```
1 export class ArrowFunction {
2   static standard() {
3     const numbers = [65, 44, 12, 4];
4     const newArr = numbers.map(function (num) {
5       return num * 10;
6     });
7     return newArr;
8   }
9
10  static modified() {
11    const numbers = [65, 44, 12, 4];
12    const newArr = numbers.map((num) => num * 10);
13    return newArr;
14  }
15 }
```

Listing 7.11 – Arrow function smell

```
export class BinaryLiterals {
  static standard() {
    const binaryValue = parseInt("1111000101", 2);
    return binaryValue == 965;
  }

  static modified() {
    const binaryValue = 0b1111000101;
    return binaryValue == 965;
  }
}
```

Listing 7.12 – Binary literals smell

```
export class InlineFunction {
  static standard() {
    const sqrt = function (value) {
      return value * value;
    };
    return sqrt;
  }

  static modified() {
    return (value) => value * value;
  }
}
```

Listing 7.13 – Inline function smell

```
export class Iterator {
  static standard() {
    const arr = [1, 22, 13, 1, -2, 3, 6, 0.78];
    let sum = 0;
    for (var i = 0; i < arr.length; i++) {
      sum += arr[i];
    }
    return sum;
  }

  static modified() {
    const arr = [1, 22, 13, 1, -2, 3, 6, 0.78];
    let sum = 0;
    for (let value of arr) {
      sum += value;
```

```
    }
    return sum;
  }
}
```

Listing 7.14 – Iterator smell

```
export class LongParameter {
  static standard() {
    return (x1, y1, x2, y2) => {
      return Math.sqrt(Math.pow(x1 - x2, 2) + Math.pow(y1 - y2, 2));
    };
  }

  static modified() {
    return (p1, p2) => {
      return Math.sqrt(Math.pow(p1.x - p2.x, 2) + Math.pow(p1.y - p2.y, 2));
    };
  }
}
```

Listing 7.15 – Long parameter smell

```
export class Looping {
  static standard() {
    var d = 35;
    var math_sind = Math.sin(d) * 10;
    var y = 0;
    for (var i = 0; i < 10; i++) {
      y += math_sind;
    }
    return y;
  }

  static modified() {
    const d = 35;
    const math_sind = Math.sin(d) * 10;
    let y = 0;
    let i = 0;
    do {
      y += math_sind;
      i++;
```

```
    } while (i < 10);
    return y;
  }
}
```

Listing 7.16 – Loop unrolling smell

```
export class MapTest {
  static standard() {
    const map = [];
    map["a"] = 10;
    map["b"] = "works";
    map["c"] = 0.47;
    return "a" in map;
  }

  static modified() {
    const map = new Map();
    map.set("a", 10);
    map.set("b", "works");
    map.set("c", 0.47);
    return map.has("a");
  }
}
```

Listing 7.17 – Map smell

```
export class TemplateString {
  static standard() {
    const name = "Philip";
    const job = "professor";
    const age = 32;
    let sentence = name + " is a " + job + "She is " + age + "
     years old.";

    return sentence;
  }

  static modified() {
    const name = "Philip";
    const job = "professor";
    const age = 32;
    let sentence = ‘ {name} is a {job}. He is {age} years old.‘;
    return sentence;
```

```
17     }
18 }
```

Listing 7.18 – Template string smell

APPENDIX B: SOURCE CODE ANALYZES OSTRICH BENCHMARK

	Nqueens		Crc		Lud	
Metric	C	JS	C	JavaScript	C	JavaScript
Program length	2157	1640	761	4721	1538	2398
Program vocabulary	149	127	132	2148	409	407
Estimated length	963,83	761,25	821,22	23433,2	3386,52	3280,16
Purity ratio	0,45	0,46	1,08	4,96	2,2	1,37
Volume	15571,75	11461,44	5360,78	52255,7	13343,62	20787,99
Difficulty	181,88	245,45	85,69	26,92	50,04	79,51
Program effort (10^5)	28,32	28,13	4,59	14,07	6,68	16,53
Time to program (h)	43,71	43,41	7,09	21,71	7,09	25,51
Cyclomatic complexity	31,5	32,8	32,7	12	49,5	14,7

	Nw		Bfs		Hmm	
Metric	C	JS	C	JavaScript	C	JavaScript
Program length	4331	3286	1350	983	4865	3560
Program vocabulary	211	163	118	121	225	180
Estimated length	1492,38	1038,08	707,44	717,19	1615,38	1195,1
Purity ratio	0,34	0,32	0,52	0,73	0,33	0,34
Volume	33440,08	24147,92	9291,57	6801,24	38014,05	26671
Difficulty	279,6	407,28	163,54	131,37	279,81	256,54
Program effort (10^5)	93,50	98,35	15,20	8,94	106,	68,42
Time to program (h)	144,29	151,78	23,45	13,79	164,15	105,59
Cyclomatic complexity	28,5	11	39,5	33,5	17,4	11

	Page-rank		Lavand		Spmv	
Metric	C	JS	C	JavaScript	C	JavaScript
Program length	11224	10863	2197	1715	545	1820
Program vocabulary	1601	1574	175	145	117	196
Estimated length	16788,39	16362,62	1192,96	909,31	707,74	1322,01
Purity ratio	1,5	1,51	0,54	0,53	1,3	0,73
Volume	119476,76	115367,45	16370,31	12313,54	3744,35	13858,77
Difficulty	126,9	99,82	128,38	158,27	60,69	138,81
Program effort (10^5)	1516,12	115,15	21,02	19,49	2,27	19,24
Time to program (h)	233,97	177,71	32,43	30,07	3,51	3,51
Cyclomatic complexity	21,8	10,4	73,3	9,3	28	10,5

	Fft		Srad		Back-propagation	
Metric	C	JS	C	JavaScript	C	JavaScript
Program length	1395	1102	1507	1101	1970	1333
Program vocabulary	106	109	127	121	145	152
Estimated length	633,17	632,91	798,75	722,76	942,69	967,24
Purity ratio	0,45	0,57	0,53	0,66	0,48	0,73
Volume	9385,45	7458,54	10531,95	7617,67	14144,42	9661,49
Difficulty	117,49	118,16	105,9	115,71	138,3	108,06
Program effort (10^5)	11,03	8,81	11,15	8,81	19,56	10,44
Time to program (h)	17,02	13,6	17,21	13,6	30,19	16,11
Cyclomatic complexity	14	10,2	143	11	15,1	9,5