Improving Developer Productivity on Internet of Things using JavaScript

Fernando L. Oliveira[1][a], Rafael R. Parizi[1][b] and Júlio C. B. Mattos[2][c]
[1]*Lardev Research Group, Federal Institute Farroupilha, 305 Otaviano Mendes st, São Borja, Brazil*
[2]*Technological Development Center – CDTec, 373 Gonçalves Chaves St., Pelotas, Brazil*

Keywords: Internet of Things, Embedded Software, JavaScript.

Abstract: C is a compiled language traditionally used to develop Internet of Things (IoT) systems. It requires higher target domain knowledge and attention to manual issues like memory management, particularly on constrained devices. In addition, the growing complexity of applications has fostered the use of interpreted languages for programming embedded software. However, little is known about how interpreted languages improve the development of IoT software. This paper reports an experiment comparing JavaScript and C languages over performance and coding. We implemented solutions for the same problem at hand through each language, keeping the same hardware platform. As a result, we identified that the JavaScript language could be considered an alternative for the Design Space Exploration phase. Since the perceived benefits from the programmer perspective overcome the higher performance achieved in the C-based solution, collaborating to better understand the trade-off between development, maintainability, and optimization on constrained devices.

1 INTRODUCTION

Over the years, the Internet of Things (IoT) has become pervasive in our daily routine. It has supported innovation in the industry, agriculture, transport, commerce, health, home automation, to name but a few. According to Transforma Insights (Transforma Insights, 2021), the forecast until the year 2030 will be a total of 24.1 billion active IoT devices, with a growth rate of 11% per year.

Commonly IoT devices are constrained in terms of processing, memory, storage, and power consumption. These characteristics imply a software development process that must consider resource limitations. Hence, developers are forced, except for a few cases, to build specific hardware platform-based solutions (Liggesmeyer and Trapp, 2009).

Traditionally, embedded software development uses compiled programming languages such as C language. C dominates constrained devices (Ebert and Jones, 2009), (Severin et al., 2020), (Eclipse Foundation, 2020); it is also a natural choice for embedded coding because most microcontrollers support it. Also, it is the only option for some devices due to their constrained properties. Moreover, the C compiler introduces optimizations (interprocedural optimizations) to achieve performance, reduce the binary footprint, and best manage resources (Oshana and Kraeling, 2019).

However, the increasing complexity of embedded software imposes requirements on the software design. It needs to attend non exclusively resource management but also consider the aspects of software development, such as maintainability, reuse, and testability (Papadopoulos et al., 2018), (Oliveira et al., 2008). Therefore, the C language can be unsuitable considering the language structure and the lack of utility functions. For instance, pointers management might have difficulty understanding the logic implementation, and memory management need extra attention to avoid memory leaks.

At the same time, the hardware evolution allows the introduction of new approaches, techniques, and technologies. Thus, the Design Space Exploration (DSE) can explore different design alternatives for modeling and planning applications and standards (Thole and Ramu, 2020). Therefore, DSE for embedded systems explores the design space to make choices about which technologies should be used and consider optimization objectives and software engineering metrics (Pimentel, 2016).

In this paper, we introduce JavaScript (JS) lan-

[a] https://orcid.org/0000-0002-9158-8879
[b] https://orcid.org/0000-0001-8550-1135
[c] https://orcid.org/0000-0002-0619-9271

Oliveira, F., Parizi, R. and Mattos, J.
Improving Developer Productivity on Internet of Things using JavaScript.
DOI: 10.5220/0011082200003194
In *Proceedings of the 7th International Conference on Internet of Things, Big Data and Security (IoTBDS 2022)*, pages 223-230
ISBN: 978-989-758-564-7; ISSN: 2184-4976
Copyright © 2022 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved

guage as an alternative to a programming language that can be considered at the DSE phase. We developed two versions of the same application using C and JavaScript language. Both solutions were deployed in the same hardware platform (ESP8266 NodeMCU-12E (ESP8266 NoDEMCU-12E, 2021)). We extracted data about execution and resource consumption. Also, we performed analyses from the source code using software quality metrics. Our findings show that although the C language has better performance than JavaScript, the developer perspective's benefits can overcome the poor performance. Furthermore, we identified that the JS language could be a key to creating a cross-platform development environment.

The outline of this paper is as follows: Section 2 presents the essential concepts about source code analysis in a nutshell; Section 3 describes the methodological process of the research; Section 4 shows the results and discussions, and finally, Section 5 concludes this paper and gives an outlook on future work.

2 BACKGROUND

The search for application development improvements has been investigated in detail by the researchers. There are many techniques, approaches, and methodologies to contribute, somehow, toward better software development.

Choosing of programming language is one of the most complex decision-making that the software engineer must take (Bhattacharya and Neamtiu, 2011). Several items could impact the decision process, such as implementation cost, quality of result, learning curve, maintainability, language ecosystem, among others. Therefore, each language has different traits that should be considered in selecting a fitting language (Farshidi et al., 2021).

Thereby the software engineering concerns of software product quality to achieve software requirements. Thus, the programming language features have a strong impact on the programming attributes like usability, cost, product, supplies, and maintainability. Then knowledge about programming languages assumes an essential role in the developer process, in particular, in the embedded system domain (Farshidi et al., 2021).

Given the context, source code analysis aims to seek code quality. Nevertheless, the analysis can be performed considering distinct goals; for instance, optimization, maintenance, and reuse are interested in software engineering perspective, even though complexity, code size, and available resource are needs from the environment or device (Mushtaq et al., 2017).

Therefore, the source code analysis may take statically, dynamically, or combined using tools. Moreover, the static analysis evaluates the raw source code to extract information about the program (Binkley, 2007). These analyses help the developers detect logical defects, better understand the program flow, minimize security issues, and measure code complexity, among others.

The Halstead Metrics (Halstead, 1977) is one of the classic and widely used tools for software quality metrics. It was proposed by Maurice Halstead and comprehends software complexity measures based on the numbers of operators and operands from source code. In this way, it is possible to assess different languages and use them as a quality parameter to compare them.

Static code analysis allows us to perform statistical analysis to evaluate and compare programming languages against each other. Likewise, the empirical study of programming language could evidence factors that led the designers to adopt or not determined language, helping in the decision making about which the best fitting programming language for one specific scenario (Meyerovich and Rabkin, 2013).

2.1 Related Work

The overwhelming majority of research from literature addresses code analysis over the desktop, server, or web perspective. We have been selected works that, in some way, address code analysis from the embedded systems outlook.

Prashant and Gurumurthy (Joshi and Gurumurthy, 2014) investigate optimizations at the source code level to ARM processors in the embedded system context. Their research addresses loop transformation techniques; Their results point to a gain of 40% concerning the code density and by 30% in the speed of operation.

Kienle et al. (Kienle et al., 2012) explore static code analysis up a specific system. They presented a case study of an industrial embedded system developed with C language. As a result, the authors reported insights that can be used as a complement to more generic code analysis tools.

Otherwise, Oliveira et al. (Oliveira et al., 2008) investigate the relationship between quality metrics for software products and physical metrics for embedded systems. They used Java language and concluded a strong correlation between quality metrics for traditional software products and performance metrics for embedded systems.

Commonly, code analysis approaches for embedded systems concern the devices' performance and sometimes overlook the software quality metrics. Therefore, it is essential to evaluate the impact over source code apart from performance considering other panoramas such as the developer or software engineering perspective.

We explore JavaScript as an alternative to programming language in the Internet of Things context. Furthermore, to the best of our knowledge, this is the first evaluation of the JavaScript language over-development viewpoint applied in the embedded software domain. In particular, we investigate the advantages and disadvantages (focusing on constrained devices) of JS against C language regarding resource consumption and software quality metrics such as complexity, maintenance, and reuse.

3 METHODOLOGY

We developed two versions of the same application using C and JavaScript language. The application consists of home automation integrated with the Google Smart Home platform (Google Actions, 2021). It allows users to control their devices through the Google Home app, Google Assistant (via voice command), and custom applications. In particular, this automation proposal controls an automatic garage door opener through 433MHz frequency.

Each command from the server to the IoT device is represented as a JSON object (JavaScript Object Notation) that describes the request, device, and action to perform. This object is called Google HomeGraph. Furthermore, we included an extra parameter (RFCode) inside the model, which contains the binary code to interact with the garage opener. Finally, we collected the RF code to open and close the door through the RF receiver sensor.

Regard the hardware, we elected the ESP8266 NodeMCU-12E (NodeMCU) (ESP8266 NoDEMCU-12E, 2021) as the microcontroller. It is a popular development board integrated with a Wi-Fi communication chip, 80MHz CPU frequency, 64kB of memory, and 4MB for storage. According to the RFC 7228 (RFC 7228: Terminology for constrained-node network, 2021) parameters, it is considered a constrained IoT device. In addition, we use the FS1000A 433MHz RF transmitter (Components Info, 2021) for wireless communication between the IoT device and embedded garage door circuit. Figure 1 presents the application flow for the proposed application.

Figure 1 shows that the user can start the interaction by saying a command. It is processed at Google

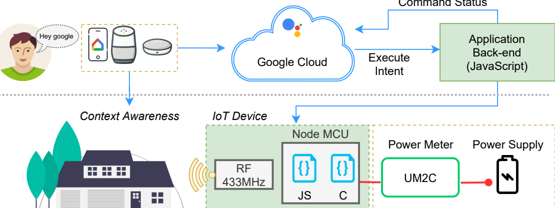


Figure 1: Application flow.

cloud, which finds an intent and sent to the application server, and finally, it notifies the edge device. This process could be started without voice command using the Google Home app or any in-house application. If the user has Google devices in the same network, the command cannot need to be processed on the cloud server because these devices can load and run custom applications (written in JS) inside upon. From that, the communication is over Wi-Fi, reducing the latency and increasing reliability.

To enable JavaScript on IoT devices, we need an interpreter to execute JS code. For this experiment, we selected Espruino (Espruino, 2021). Espruino is an Open Source JavaScript engine for Microcontrollers that has a full coverage of ES5[1] and partially ES6 features. Furthermore, Espruino provides firmware with support ESP8266 board and web IDE to development.

We collect execution time and memory usage from JavaScript performance API and ESP8266 SDK on JS and C algorithms. C program was compiled over Arduino IDE by default optimization settings. Regarding power consumption, we connected a multimeter in series with a battery and supplied ESP8266 from that. As a power meter, we selected the UM2C USB (UM24c USB Tester, 2021).

We performed a static code analysis from C and JavaScript source code to extract software quality metrics. For that, we selected the Halstead-Metrics-Tool (Software Engineering Research Group at Politecnico di Torino, 2021) to collect Halstead Metrics and Lizard tool (Terry Yin, 2021) to collect Cyclomatic Complexity. In addition, to insurance the evaluation approach, we apply the code analysis tools over the Ostrich Benchmark Suite (Khan, Faiz and Foley-Bourgon, Vincent and Kathrotia, Sujay and Lavoie, Erick, 2014), aiming to evaluate the results of these tools on external (neutral) codes and compare them with the results obtained from our source code to find behavior patterns.

[1]ES is an acronym for ECMAScript; This is the official name of JavaScript language used in its specification.

4 RESULTS AND DISCUSSION

Programmers can propose different solutions to the same problem. We developed the algorithms to keep the source code as simple as possible, defining only the necessary structure and importing some libraries in order to facilitate the development. This section shows the analysis from performance and developer overview.

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

4.1 Performance Analysis

The data extraction was performed from a set of executions of each algorithm. Every single request was repeated 50 times to obtain data consistency and more reliable analytical information. The results represent the average of the executions or the sum of them.

Although both algorithms were deployed over the same hardware, the experiment points out significant performance variations regarding memory consumption. Table 1 shows the memory usage details.

Table 1: Memory consumption (bytes).

Lang.	Used (%)	Free memory	Total
C	5924 (11,09)	47512	53436
JS	19880 (62,12)	12120	32000

JavaScript solution attained in huge memory consumption compared to the C algorithm. This high-memory consumption occurs because the JavaScript engine reserves memory space for the entire JavaScript heap. Also, external libraries, which typically use buffers, demand a large amount of memory (Hong and Shin, 2020).

Moreover, the available memory for each implementation is distinct as well. The C solution had $\approx$ 53kB, and at the same time, the JavaScript approach had 32kB only. That happens because part of the memory space is reserved for ESP8266 SDK to support Wi-Fi and TCP/IP protocol. In particular, the Espruino engine uses 20kB for itself plus 8kB for TCP/IP buffers.

Regarding execution time, we measured how long it took to process each request, and we clustered the results by algorithms. Figure 2 resumes performance data by language. In addition, each graph shows JavaScript results twice because Espruino sets the ESP8266 clock to 160MHz by default, but the C algorithm works over 80MHz. In this way, we decided to expose all of the data.

Section “a” in Figure 2 reports the experiment in which the 433MHz RF transmitter is used to open or close the garage door. C algorithm has taken a longer time than JS. In a deep analysis, we found that most time is spent sending RF commands through the RC-switch library. In contrast, section “b” exposes that C is, at least, four times more efficient than JavaScript. Finally, Figure 2 still reveals the data about energy consumption.

The results represent the sum of power consumption from 50 runs recorded over the whole executions. In general, JS figures out to consume more energy to develop the algorithms. It happens because the JavaScript engine performs optimizations in order to gain performance. Such analysis implies direct CPU usage, so it has an extra energy cost. The only exception is when RF commands are used, and C had the worst consumption.

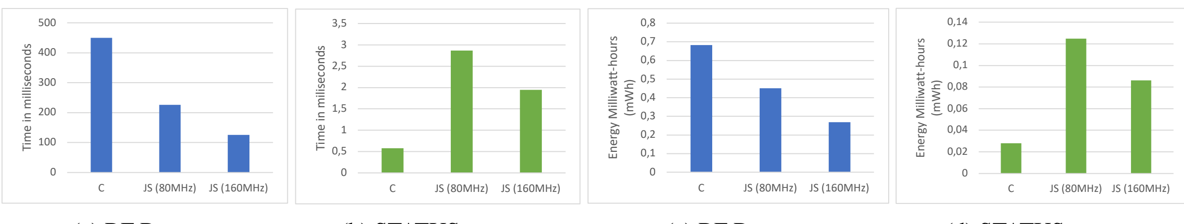
Overall, C’s performance is better than JavaScript, as it was supposed. However, the JavaScript application got a satisfactory outcome, and the decision to use it is not merely about performance issues. We consider other aspects around the developer process or associate to post-development, such as maintenance and reuse. Thus, We processed the solutions from the programmer’s perspective.

4.2 Developer Analysis

Listings 1 and 2 present chunks of the source code from both algorithms; configuration and device controls have been suppressed, keeping the focus on business logic. The full code can be found at the GitHub (The garage door, 2022).

```
1 // ...
2 void loop() {
3   server.handleClient();
4 }
5 bool isOpened() {
6   return digitalRead(magnet_switch) == HIGH;
7 }
8 void handlerStatus() {
9   if ( isValidRequest() ) {
10    const size_t bufferSize = 2 * (
      JSON_ARRAY_SIZE(2) + JSON_OBJECT_SIZE(6));
11    DynamicJsonDocument requestArgs(bufferSize);
12    deserializeJson(requestArgs, server.arg("
      plain"));
13
14    DynamicJsonDocument doc(128);
15    doc["success"] = true;
16    doc["requestId"] = requestArgs["requestId"];
```

SCIENCE AND TECHNOLOGY PUBLICATIONS



(a) RF Request (b) STATUS request (c) RF Request (d) STATUS request

Figure 2: Execution time / Energy consumption by endpoint.

```
17    doc["isOpened"] = isOpened();
18    sendResponse(doc);
19  }
20 }
21 void handlerRF() {
22  if ( isValidRequest() ) {
23    const size_t bufferSize = 2 * (
      JSON_ARRAY_SIZE(4) + JSON_OBJECT_SIZE(11));
24    DynamicJsonDocument requestArgs(bufferSize);
25    deserializeJson(requestArgs, server.arg("
      plain"));
26    DynamicJsonDocument cmd = requestArgs["inputs
      "][0]["payload"]["commands"][0]["execution"
      ][0];
27    const char* action = cmd["command"];
28    const int perc = cmd["params"]["openPercent"
      ];
29    const long RFCode = cmd["params"]["RFCode"];
30    if ( strcmp(action, "...commands.OpenClose")
      == 0   ) {
31     bool executed = false;
32     String message;
33     if ( perc == 100 ) {
34       if ( !isOpened() ) {
35         executed = performRF(RFCode);
36         message = "Opening the door";
37       } else {
38         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 1: C chunk of code.

Listing 1 represents the C implementation in which the endpoints are defined on method “hander-Status” (lines 8-20) and “handlerRF” (lines 21-56). Entry parameters and return are expected in JSON format, and for this reason, it requires manual management of the space in memory 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 one. Listing 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 2: JavaScript chunk of code.

Note that the business logic is, apparently, different between the two algorithms. For instance, the logical process from lines 30 until 47 on Listing 1 does not appear on Listing 2. This feeling occurred be-

cause the chunk of code was isolated in a common class to reuse it on other application layers. Listing 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: 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 developing adopting the Template Method design pattern, aiming the specific implementation in a separated class. This is essential to promote flexibility and contribute to interoperability among layers since each microcontroller may have distinct hardware specifications. Finally, Listing 4 show 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 4: JavaScript modeMcu class.

Listing 4 represents the custom implementation for the ESP8266 NodeMCU board. For example, if we needed to implement to another microcontroller, like Raspberry PI, we would need to create only one class to give the abstract methods, and the rest of the structure could be reused. Table 2 presents the static code analysis from the algorithms; the values inside the table represent a number on a magnitude scale according to each criterion.

Table 2: Code analysis details.

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
Cyclomatic complexity	2.1	1.8

As before illustrated, the measures shown in Table 2 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 mean the average ones. In contrast to the performance examination, JavaScript overcomes C in all items, and in some cases, the results can be almost twice:

- Program length: The size of the program;
- Program vocabulary: Number of operators and operands that composes the program;
- Estimated length: Metric of size estimated removing everything from 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: Its 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: It represents the time to develop or understand a program;
- Cyclomatic complexity: It indicates the complexity of a program based on the number of decision statements in the code, methods, and lines of code.

Regarding the cyclomatic complexity index, this measure represents how complex each code was considered. Empirical verification shows that the C algorithm got a higher complexity index than JS. Table 3 shows the detailed complexity analysis.

Table 3: Cyclomatic complexity details.

lang.	nloc[1]	token[2]	Funct. count	CCN[3]
C	102	78.6	8	2.1
JS	74	51.8	6	1.8

[1] Number of lines of code without comments;
[2] Token count of functions
[3] Cyclomatic Complexity Number

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 code analysis results[2] from the Ostrich benchmark showed that JavaScript has a balance or superior behavior; it matches with the analysis performed from code produced in this experiment, leaving us confident about the result achieved. Also, it allows us to indicate JavaScript as an alternative for the Design Space Exploration phase.

Nonetheless, IoT solutions are not simple as we proposed. But, the experiment demonstrated that JS could enhance the reuse of code and enable the build of standard components. For instance, on automation systems that need to consider different hardware sensors and actuators, JavaScript provides the tools to create solutions, frameworks, and libraries to integrate all of them, increasing the developer experience.

Regarding performance, It is strongly related to the JavaScript engine. Few engines are available on constrained devices, and the consumption of resources depends on how optimized it is. The algorithm by itself has a significant contribution to how the application performs, which can vary accord the JS engine. So programming on IoT devices is not a trivial task. We need to find the balance between development, execution, and resource consumption.

Moreover, JSON format has become a common format to transfer data inter applications. To use it on the C algorithm, we need external libraries and perform calculations to allocate the objects in a balanced way. On the other hand, JavaScript can facilitate some action, and their event-based model has a strong adherence to the IoT execution model.

JavaScript architecture for this proposal was projected to be simple and understandable, and at the same time to enhance the software quality and standardize the developer process. These approaches increase the level of abstraction and improve design productivity. Also, it is clear that software quality has a strong relation to the consumption of resources (Oliveira et al., 2008), (Papadopoulos et al., 2018).

Finally, the developing application for heterogeneous context like IoT, behind challenging, requires a simple programming model to be feasible (Krishnamurthy and Maheswaran, 2016). Also, the advance of virtual machines like Espruino, Duktape, JerryScript, and Moddable make JS a real option to programming constrained devices.

5 FINAL REMARKS

This paper evaluated whether JavaScript could be used as alternative to C language for coding on the IoT-context. We were able to develop two version of the same application and collect information about the consumption of resources such as execution time, memory, battery and analyzed the source code using software quality metrics.

We could identify that the JavaScript performance penalty is acceptable compared to the gain in abstraction and reuse in embedded system design. In addition, embedded software is not a stand-alone artifact. Thus, using the same language in all application-stack simplifies and standardizes the development process, implying positively in a more suitable environment for interoperability, reducing cost, and optimizing maintainability.

In future studies, we intend to explore which aspects of JavaScript language impact more device performance, mainly related to memory consumption. Also, we will deepen the investigation of how to JavaScript engine could affect this process since it is key to enable JavaScript as a language for the IoT environment.

ACKNOWLEDGMENT

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior -

[2]The full analysis can be found in the appendix at the GitHub repository (Paper Appendix, 2022)

Brasil (CAPES) - Finance Code 001.

REFERENCES

Bhattacharya, P. and Neamtiu, I. (2011). Assessing programming language impact on development and maintenance: A study on c and c++. In *2011 33rd International Conference on Software Engineering (ICSE)*, pages 171–180. IEEE.

Binkley, D. (2007). Source code analysis: A road map. In *Future of Software Engineering (FOSE '07)*, pages 104–119.

Components Info (2021). Fs1000a 433mhz rf transmitter. https://www.componentsinfo.com/fs1000a-433mhz-rf-transmitter-xy-mk-5v-receiver-module-explanation-pinout/. Accessed: Mar. 2022.

Ebert, C. and Jones, C. (2009). Embedded software: Facts, figures, and future. *Computer*, 42(4):42–52.

Eclipse Foundation (2020). Iot developer survey 2020. https://outreach.eclipse.foundation/eclipse-iot-developer-survey-2020. Accessed: Mar. 2022.

ESP8266 NoDEMCU-12E (2021). Espressif official documentation. https://www.espressif.com/en/support/documents/technical-documents. Accessed: Mar. 2022.

Espruino (2021). Javascript interpreter for microcontrollers. https://www.espruino.com. Accessed: Mar. 2022.

Farshidi, S., Jansen, S., and Deldar, M. (2021). A decision model for programming language ecosystem selection: Seven industry case studies. *Information and Software Technology*, 139:106640.

Google Actions (2021). Google smart home platform. https://developers.google.com/assistant/smarthome. Accessed: Mar. 2022.

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

Hong, G. and Shin, D. (2020). Segment-based multiple-base compressed addressing for flexible javascript heap allocation. *IEEE Access*, 8:185405–185415.

Joshi, P. V. and Gurumurthy, K. (2014). Analysing and improving the performance of software code for real time embedded systems. In *2014 2nd International Conference on Devices, Circuits and Systems (ICDCS)*, pages 1–5. IEEE.

Khan, Faiz and Foley-Bourgon, Vincent and Kathrotia, Sujay and Lavoie, Erick (2014). Ostrich benchmark suite.

Kienle, H. M., Kraft, J., and Nolte, T. (2012). System-specific static code analyses: a case study in the complex embedded systems domain. *Software quality journal*, 20(2):337–367.

Krishnamurthy, J. and Maheswaran, M. (2016). Chapter 5 - programming frameworks for internet of things. In Buyya, R. and Vahid Dastjerdi, A., editors, *Internet of Things*, pages 79–102. Morgan Kaufmann.

Liggesmeyer, P. and Trapp, M. (2009). Trends in embedded software engineering. *IEEE software*, 26(3):19–25.

Meyerovich, L. A. and Rabkin, A. S. (2013). Empirical analysis of programming language adoption. *SIGPLAN Not.*, 48(10):1–18.

Mushtaq, Z., Rasool, G., and Shehzad, B. (2017). Multilingual source code analysis: A systematic literature review. *IEEE Access*, 5:11307–11336.

Oliveira, M. F., Redin, R. M., Carro, L., da Cunha Lamb, L., and Wagner, F. R. (2008). Software quality metrics and their impact on embedded software. In *2008 5Th International Workshop on Model-based Methodologies for Pervasive and Embedded Software*, pages 68–77. IEEE.

Oshana, R. and Kraeling, M. (2019). *Software engineering for embedded systems: Methods, practical techniques, and applications*. Newnes.

Papadopoulos, L., Marantos, C., Digkas, G., Ampatzoglou, A., Chatzigeorgiou, A., and Soudris, D. (2018). Interrelations between software quality metrics, performance and energy consumption in embedded applications. In *Proceedings of the 21st International Workshop on software and compilers for embedded systems*, pages 62–65.

Paper Appendix (2022). Appendix a: Source code analysis from ostrich benchmark algorithms on github repository. https://github.com/fernandotetu/the-garage-door-opener/blob/main/Appendix_A.pdf. Accessed: Mar. 2022.

Pimentel, A. D. (2016). Exploring exploration: A tutorial introduction to embedded systems design space exploration. *IEEE Design & Test*, 34(1):77–90.

RFC 7228: Terminology for constrained-node network (2021). Internet engineering task force (ietf). https://tools.ietf.org/html/rfc7228. Accessed: Mar. 2022.

Severin, T., Culic, I., and Radovici, A. (2020). Enabling high-level programming languages on iot devices. In *2020 19th RoEduNet Conference: Networking in Education and Research (RoEduNet)*, pages 1–6.

Software Engineering Research Group at Politecnico di Torino (2021). Halstead metrics tool. https://github.com/SoftengPoliTo/Halstead-Metrics-Tool. Accessed: Mar. 2022.

Terry Yin (2021). Lizard cyclomatic complexity analyzer. https://github.com/terryyin/lizard. Accessed: Mar. 2022.

The garage door (2022). Github repository. https://github.com/fernandotetu/the-garage-door-opener. Accessed: Mar. 2022.

Thole, S. P. and Ramu, P. (2020). Design space exploration and optimization using self-organizing maps. *Structural and Multidisciplinary Optimization*, 62(3):1071–1088.

Transforma Insights (2021). The internet of things (iot) market 2019-2030. https://transformainsights.com/news/iot-market-24-billion-usd15-trillion-revenue-2030. Accessed: Mar. 2022.

UM24c USB Tester (2021). Hangzhou ruideng technologies. https://rdtech.en.alibaba.com/. Accessed: Mar. 2022.