MoMIT: Porting a JavaScript Interpreter on a Quarter Coin

Rodrigo Morales, *Member, IEEE,*
Rubén Saborido, *Member, IEEE,*
and Yann-Gaël Guéhéneuc, *Senior Member, IEEE*

Abstract—The Internet of Things (IoT) is a network of physical, connected devices providing services through private networks and the Internet. The devices connect through the Internet to Web servers and other devices. One of the popular programming languages for communicating Web pages and Web apps is JavaScript (JS). Hence, the devices would benefit from JS apps. However, porting JS apps to the many IoT devices, *e.g.,* System-on-a-Chip (SoCs) devices (*e.g.,* Arduino Uno), is challenging because of their limited memory, storage, and CPU capabilities. Also, some devices may lack hardware/software capabilities for running JS apps "as is". Thus, we propose *MoMIT*, a multiobjective optimization approach to miniaturize JS apps to run on IoT devices. We implement *MoMIT* using three different search algorithms. We miniaturize a JS interpreter and measure the characteristics of 23 apps before/after applying *MoMIT*. We find reductions of code size, memory usage, and CPU time of 31%, 56%, and 36%, respectively (medians). We show that *MoMIT* allows apps to run on up to two additional devices in comparison to the original JS interpreter.

Index Terms—Internet of Things, Software Miniaturization, Multiobjective optimization, embedded devices, JavaScript, Evolutionary algorithms

1 Introduction

The Internet of Things (IoT) is a network of physical, connected devices providing services [1] through private networks and the Internet. In 2016, Gartner[1] predicted that 75% of Internet of Things (IoT) projects will take up to twice as long as planned in 2018 due to insufficient staffing/expertise. Meanwhile, more companies want to seize the business opportunities offered by the IoT. They must propose Web services and Web apps on different architectures to reach as much customers as possible. Thus, they must spend time and effort developing/migrating and maintaining their apps on different devices. However, they face the large diversity of hardware and software, from Cloud virtual machines to Systems-on-a-Chip (SoCs), and need help to map/remove features from their apps, to deploy them on constrained devices.

Today's programming language for Web pages and Web apps is JavaScript (JS). JS is among the most popular programming languages in the last five years [2]. It is commonly used to develop event-driven IoT systems. It can handle large networks of devices and perform asynchronous computations, *i.e.,* a typical scenario with apps. It powers many IoT projects, including some of IBM and Samsung. Using the same JS app on many devices would help these companies deploy on their diverse hardware and software. However, there is a shortage of JS developers with IoT experience [3], [4] and memory, storage, and CPU constraints prevent the direct use of JS.

• *R. Morales, R. Saborido, Y.-G. Guéhéneuc are with Concordia University Montréal, Québec, Canada. E-mail: rodrigo-morales2@acm.org, ruben.saborido-infantes@concordia.ca, yann-gael.gueheneuc@concordia.ca.*

Manuscript received March 2019.

1. http://tiny.cc/v36fdz

Objective: We want an approach to *miniaturize* JS interpreters to run Web apps on diverse hardware, including, constrained devices, by removing from the interpreters the code features unneeded to run the apps. Table 1 shows examples of such features. Section 5 describes the experiments that we run to find these most impactful features.

Context: A company developed some apps in JS on regular hardware. It now wants to run it on constrained devices. It could either translate its app into C or compile/bundle a JS interpreter with the app. The first scenario implies the cost of maintaining two apps in C and JS in parallel while the second only requires removing unnecessary features from the interpreter. We choose the second scenario.

Method: We formulate the problem of miniaturizing an interpreter as a multiobjective optimization problem considering three objectives: memory, storage, and CPU. Given a JS interpreter, we propose *MoMIT*, an approach taking as input (1) a set of available code features F; (2) a list L of compulsory features $ComF \subset F$; and, (3) a list of IoT devices specifications. *MoMIT* finds the combinations of optional and compulsory features that best satisfy the devices' constraints for each objective.

Results: We design and implement *MoMIT* as a multiobjective miniaturization approach that combines the configuration options of a JS interpreter to run an unmodified JS app. We apply *MoMIT* on the *Duktape* JS interpreter. We identify 86 features among 283 that companies can (de)activate to miniaturize *Duktape* and run their apps on constrained devices. We reveal 10 hidden dependencies within 20 of the 86 features of *Duktape* that prevented miniaturization. We detect and report a bug in *Duktape* preventing developer to customize their interpreter. (The bug was fixed and the corresponding patch committed to the repository's

master branch (issue #1990[2]). We conduct a comprehensive empirical study on 23 JS tests of a benchmark for Web browsers and show that *MoMIT* allows running them on many devices, **up to the size of a quarter coin**. We compare three different algorithms, *i.e.,* NSGA-II, *RS+*, and SWAY in terms of execution time and quality of their solutions to show that *MoMIT* is independent of the search algorithm. We release *MoMIT* as open source along with the outcome data result of our experiments for the IoT community.

Although we focus on miniaturizing the *Duktape* JS interpreter to constrained devices, *MoMIT* is general and supports other code interpreters, *e.g.,* Lua, Python, etc.

The Remainder of this paper is organized as dollows. Section 2 relates our work to the state of the art. Section 3 provides foundations on multiobjective optimization, evolutionary algorithms, IoT systems, JS, and software miniaturization. Section 4, presents our automated multiobjective approach for miniaturizing JS engines, *MoMIT*. Section 5 reports a preliminary study of the impact of JS features on performance metrics. Section 6 describes the implementation of *MoMIT* with different evolutionary algorithms. Section 7 summarizes the experimental setting for evaluating *MoMIT*. Section 8 provides the results of our experiments while Section 9 discusses them and Section 10 threats to their validity. Section 11 concludes with future work.

2 Related Work

We present related work divided in four categories. To the best of our knowledge, we are the first to address the problem of miniaturizing interpreters for IoT using a multiobjective approach.

2.1 Programming Language Migration

Software migration is developers porting source code written in one programming language into another. It is tedious and error-prone and requires developers to define manually migration rules between the origin and target language constructs, including between (non-)equivalent APIs interfaces of third-party libraries. (Semi-)automatic tools were proposed to ease migration, *e.g.,* Mossienko et al. [5] proposed an approach to migrate COBOL to C and Sharpen [3] allows developers to migrate Java to C#. Other tools were developed for migrating Web sites to modern APIs [6], [7]. These tools require developers to define migration rules to customize and perfect their conversion. There is no tool support available for migrating JS to C, which is the main programming language for IoT devices.

2.2 Software Product Lines

A software-product line (SPL) defines a set of software sharing code among them [8]. From one software-product line, several software can be generated. Siegmund et al. [9] modelled the compilation configuration options of database systems as a product line. By turning on/off these options, different databases could be generated. However, the more options, the more difficult is the search for valid combinations of these options. Turning *on* or *off* features randomly has a low probability of satisfying the constraints of a valid software. We corroborate this difficulty when implementing a *pure random search* to test *MoMIT*, which returns no valid software among 250 candidates, due to dependency among features discussed in Section 8. Recent works, *e.g.,* Sayyad et al. [10] and Chen et al. [11] combined metaheuristics with a preprocessor (a SAT solver) to reduce the search-space to a subset of valid candidates instead of randomly generating and evaluating candidates.

2.3 Compilers Optimization

Compiler options is an effective way to increase the quality of executable code. Modern compilers can compile for many hardware/software and implement many optimizations, which are not always appropriate for a given target. Hossein et al. [12] evaluated different autotuning approaches to choose compiler options and the ordering of their optimizations. They demonstrated that these approaches have positive effects on the performance of apps, with up to 60% improvements. Souza and Silva [13] presented a design-space exploration approach to search for a compiler optimization sequence. Their approach relies on sequences previously generated for a set of training apps. They showed that optimized sequences generated code outperforming the standard optimization level O3 by an average improvement of 7% on two benchmark suites. Plotnikov et al. [14] presented a tool for automatic compiler tuning to improve the performance of several popular apps, including GCC itself. Luque et al. [15] used parallel metaheuristic techniques to choose compiler options when compiling a set of apps and to improve their runtime performances. Georgiou et al. [16] observed that fewer optimizations can yield significant savings in runtime performance and energy consumption.

2.4 Software Miniaturization

Software Miniaturization was introduced by Di Penta et al. [17] to reduce the footprint of software during its porting to hand-held devices. Their approach helps removing dead code, refactoring code clones, and eliminating circular dependencies. They evaluated their approach on a geographic information-system and reduced the average number of objects by 50%. Ali et al. [18] proposed a multiobjective approach to miniaturize apps based on customers' prerequisites, storage occupation, and CPU usage. They applied their approach on an email client and an instant messenger and showed a reduction of the manual effort by 77% on average. We choose a different approach miniaturizing the interpreter that executes an apps, not the app itself. We do not modify the app. Both approaches could be applied on the same app to reduce its footprint even more.

3 Background

This section provides a background on multiobjective optimization and the IoT to understand our approach.

3.1 Multiobjective Optimization

The general formulation of a *multiobjective optimization problem* is given by:

2. https://github.com/svaarala/duktape/issues/1990
3. https://github.com/mono/sharpen

Table 1: JS interpreter most-impactful features for miniaturization to IoT devices

ID	Name	Description	Value	Code Size δ	Mem. Usa. δ	CPU Time δ
3	DUK_USE_EXEC_PREFER_SIZE	Prefer size over performance in bytecode executor	TRUE	-11.71	0	9.15
7–10	Divers (Store objects in ROM)	built-ins for compiling objects and strings as constants in the ROM space to reduce RAM usage at the cost of a larger code footprint and slower performance	TRUE	25.21	-87.93	-13.41
86	DUK_USE_REGEXP_CANON_ WORKAROUND	Use a 128kB lookup table for improving RegExp processing performance at the cost of code size	FALSE	23.59	0	-14.63
5	DUK_USE_LIGHTFUNC_BUILTINS	Force built-in functions to be lightweight functions. This reduces memory footprint by around 14 kB at the cost of some non-compliant behavior.	TRUE	0	-37.43	-12.805
43	DUK_USE_BUFFEROBJECT_SUPPORT	Enable support for Khronos/ES 6 typed arrays and Node.js Buffer objects. This includes all ArrayBuffer, typed array, and Node.js Buffer methods.	FALSE	-4.18	-21.39	-14.02
19	DUK_USE_DATE_BUILTIN	Provide a Date built-in	FALSE	-0.11	-11.13	-13.41
11	DUK_USE_REFERENCE_COUNTING	Use Automatic Reference Counting for Memory management to remove objects that are no longer needed	FALSE	-6.79	10.62	-12.2
84	DUK_USE_FASTINT	Enable support for 48-bit signed "fastint" integer values. Fastints are transparent to user code (both C and Ecmascript) but may be faster than IEEE doubles on some platforms. The downside of fastints is increased code footprint and a small performance penalty for some kinds of code.	TRUE	6.67	0	-23.17
15	DUK_USE_ARRAY_BUILTIN	Provide an Array built-in.	FALSE	-1.7	-4.85	-15.85
6	DUK_USE_PREFER_SIZE	Catch-all flag which can be used to choose between variant algorithms where a speed-size tradeoff exists (e.g. lookup tables).	TRUE	-0.76	-0.03	94.51

$$\begin{array}{ll} \text{minimize} & \{f_1(\mathbf{x}), f_2(\mathbf{x}), \ldots, f_m(\mathbf{x})\} \\ \text{subject to} & \mathbf{x} \in S, \end{array} \quad (1)$$

where m ($m \geq 2$) and the *objective functions* $f_i : \mathbb{R}^n \rightarrow \mathbb{R}$ ($i = 1, \ldots, m$) must be minimized and $S \subset \mathbb{R}^n$ is the *feasible set*. A decision vector $\mathbf{x} = (x_1, \ldots, x_n)^T$ is a *feasible solution* if it belongs to S. Its image $\mathbf{z} = \mathbf{f}(\mathbf{x}) = (f_1(\mathbf{x}), \ldots, f_m(\mathbf{x}))^T$ is an *objective vector* and the set of all objective vectors, denoted as $Z = \mathbf{f}(S) \subset \mathbb{R}^m$, is the *feasible objective set*.

Conflicts among objectives make it impossible to find a feasible solution that simultaneously minimizes all objectives. Given $\mathbf{z}, \mathbf{z}' \in \mathbb{R}^m$, we say that $\mathbf{z}$ *dominates* $\mathbf{z}'$ if $z_i \leq z'_i$ for all $i = 1, \ldots, m$ and $z_j < z'_j$ for, at least, one index j. When $\mathbf{z}$ and $\mathbf{z}'$ do not dominate each other, we say that they are *non-dominating*. For problem (1), a *Pareto optimal solution* is a feasible solution $\mathbf{x} \in S$ for which there does not exist another $\mathbf{x}' \in S$ such that $\mathbf{f}(\mathbf{x}')$ dominates $\mathbf{f}(\mathbf{x})$. The set of all Pareto optimal solutions, E, in the decision space, is the *Pareto optimal set* and its image in the objective space, $\mathbf{f}(E)$, is the *Pareto optimal front* (PF).

3.2 Multiobjective Evolutionary Algorithms

Evolutionary multiobjective Optimization (EMO) algorithms can solve multiobjective optimization problems [19], [20], [21], [22], [23], [24]. They find a subset of non-dominated solutions approximating PF. The approximation set is composed of solutions as evenly distributed as possible in PF (diversity) and as close as possible to the true PF (convergence). In particular, NSGA-II [25] is an EMO algorithm that has solved many real-life multiobjective optimization problems [20], [26]. It uses an elite-preserving strategy and a diversity preserving mechanism and uses a fast non-dominated sorting procedure to rank the solutions into several non-dominated fronts to select the best individuals.

3.3 Breadth and Depth of the IoT

The IoT involves a range of key topic [27], including: hardware, networking, software design, software development, security, business intelligence, data analytics, machine learning, and artificial intelligence. We focus specifically on hardware and software development.

Hardware: An IoT *device* is any piece of hardware designed/adapted to perform a particular task. In this work, we consider off-the-shell boards, which we divide in two categories: single-board computers (SBCs) and systems-on-chips (SoCs). Generally, IoT devices are characterized by their dimensions, processing power, memory, storage capabilities, and connectivity. Table 2 presents three of the most popular SoCs (Rows 1–3) and two SBCs (rows 4–5). It is not exhaustive but a selection of the most relevant IoT devices available at the time of writing according to IBM [28]. Column "Cloud enabled" indicates if the device includes pre-integrated cloud platform to manage a set of IoT devices, only available for *Particle* devices. We did not include *Arduino* devices because they target hobbyists rather than industrial systems, *e.g.,* the *Arduino UNO* offers only 2 KB of memory and 32 KB of storage, does not include Wi-Fi, and cost more than both *Photon* and *ESP32*.

Software: Developers usually use cross-platform IDEs to mitigate the burden of developing/using device-specific libraries for each different targeted device. They use the MQTT protocol or Web sockets to connect devices to edge nodes/Cloud servers and perform data analytics on the data collected by the devices. They usually develop the front-ends of their apps using JS and, thus, would benefit of using JS *also* on the devices to reduce the complexity and challenges of managing a multi-language app. However, the standard programming language for developing apps for IoT devices is C/C++. JavaScript and Python are also possible through *Tessel*, *Particle.io*, *MicroPython*, or *WeIO* but developers then must adapt manually their apps to the constraints of each device (memory, storage, and CPU).

3.4 JS Engines for IoT Devices

Moreover, because JS is a high-level interpreted programming language, developers often cannot deploy a full JS interpreter on their IoT devices because of their limited resources. One solution is to use a device that can run JS code natively, *e.g.,* Espruino, although it is expansive and uncommon in industrial settings. Alternatively, JS engines exist that can produce lightweight JS interpreters to be deployed in highly-constrained devices: *Duktape* [29],

Table 2: Comparison of the IoT devices used in this work

Name	Processor	Memory (KB)	Storage (KB)	Wi-Fi	Dimensions (mm)	Weight (g)	Cloud enabled	Price (US)
Photon	STM32 ARM Cortex M3	128	1,000	yes	36.58 x 20.32 x 4.32	5	yes	19.00
ESP32	XTENSA DUAL-CORE-32-BIT	512	4,000	yes	55.3 x 28.0 x 12.3	9.6	no	19.95
JN5168	*JN5168* 32 bit RISC microprocessor	32	256	no	24.5 x 30.5 x 9.77	4	no	26.95
RPI 3 Model B+	ARM Cortex-A53 CPU	1,000,000	16,000,000	yes	85 x 56 x 1.6	42	no	54.40
BeagleBone Black	AM3358 ARM Cortex-A8	256,000	4,000,000	yes	86.40 x 53.3	39.68	no	89.00

tinyJS[4], or JerryScript[5]). These engines allow developers to enable/disable the code features and interpreter parameters to execute their apps on constrained devices. However, developers must explore manually the space of all possible combinations of features, parameters, and constraints.

4 APPROACH

We introduce *MoMIT* (Multiobjective Software Miniaturization for the Internet of Things) to help developers port a JS interpreter to some IoT devices. We describe *MoMIT* as if applied by a company without any assumption on the available tool support, described in Section 6.

4.1 Pre-requirement Elicitation

Pre-requirement elicitation identifies the set of pre-requirements (PRs), including customers' expectations, required features, etc. PRs can be identified by developers or using static code-analysis tools, *e.g.,* JSAnalyse[6].

4.2 Selection of IoT Device Candidates

Developers must decide the IoT devices on which to deploy their apps. They must provide devices specifications, *e.g.,* available memory, and order the devices by preference for *MoMIT* to prioritize them.

4.3 Feature Identification

Let F be the set of all features provided by a JS interpreter. Basic features of the interpreter, *e.g.,* primitive types, cannot be disabled and are excluded from F. Let $ComF \subset F$ be the compulsory features required by the company that the interpreter must provide. These are features necessary for the correct execution of the app and compliance with JS standards, *e.g.,* ECMAScript (ES).

Let $DepF$ be the dependencies among features: $f_i \in F$ is dependent on $f_j \in F$ with $i \neq j$, if f_i requires f_j to have a value $v \in \{false, true\}$ to produce a valid JS interpreter. A valid interpreter is a customized interpreter that was successfully build and run and that provides $ComF$ plus a subset of F while satisfying $DepF$.

Developers must map PRs with the features F of the interpreter and give to *MoMIT* F, $ComF$, and $DepF$.

4. http://tinyjs.net
5. http://jerryscript.net
6. https://archive.codeplex.com/?p=jsanalyse

4.4 Selection of Feature Combinations

MoMIT identifies the set of features satisfying the constraints imposed by the IoT devices specifications. It starts with the compulsory features $ComF$ and finds sets of optional features $OF \equiv \{g_1, \ldots, g_N\}$ through multiobjective optimization, with N the number of optional features and $g_i \in F$ with $i = \{1, \ldots, N\}$. A miniaturized interpreter can implement the optional features $F' \subset OF$. There exist 2^{OF} possible sets F'.

MoMIT considers that an interpreter divides into M implementation units $IU \equiv \{iu_1, iu_2 \ldots, iu_M\}$. Function $Impl$ takes as input a set of features and returns the corresponding implementation units. A miniaturized interpreter is $IU' = Impl(F' \cup ComF)$.

Including/excluding a feature requires dealing with a set of property values $P \subset \mathbb{R}^K$, with K the number of property values, and with a set of internal constraints $IC \equiv \{ic_1, ic_2, \ldots, ic_K\}$, each of them imposing a set ic_j of acceptable values on the corresponding property values, with $P \in IC \equiv \{p_j \in ic_j \forall j = 1, \ldots, K\}$. Function $Prop_j(IU')$ returns the property value of an interpreter with respect to constraint j with $j \in IC$.

For the sake of simplicity, we focus on memory usage, storage usage, and CPU time, although other constraints could be considered, *e.g.,* network connectivity, energy consumption, form factor, etc. We represent the set of potential IoT devices to port an interpreter as $L \equiv \{l_1, l_2 \ldots, l_l\}$.

To measure the extent to which a miniaturized interpreter IU' matches the constraints of a device l, we define the device-satisfaction rate of l in Equation 2:

$$DSR_l(IU') = \frac{\sum_{j=1}^{K} \frac{Prop_j(IU') - ic_j(l)}{ic_j(l)}}{K} \quad (2)$$

A company would rank each device i according to some criteria, *e.g.,* its customers' preferences: val_i, where $1 \leq val_i \leq V_{max}$ and $Val \equiv \{val_1, val_2, \ldots, val_L\}$. We then define a satisfaction measure, the customer's satisfaction rate (CSR) in Equation 3, expressing the extent to which a miniaturized interpreter matches these preferences.

$$CSR(IU') = \frac{\sum_{i=1}^{L} DSR_i(IU') \times \frac{val_i}{Vmax}}{L} \quad (3)$$

Equation 4 shows the equation describing the miniaturization problem, which *MoMIT* solves using some search algorithm to obtain optimal combinations of features, which are miniaturized interpreters that (1) maximize customer's preferences CSR, *i.e.,* minimize $-CSR$, and (2) satisfy the constraints IC by minimizing $Prop(IU') \in IC$. *MoMIT* is independent of the search technique. Any search algorithm could be use to find the best combination of features.

$$\min_{F' \in 2^{OF}} (-CSR(IU'), Prop(IU')) \quad (4)$$

We formulate the problem of miniaturization as a multiobjective problem and, thus, *MoMIT* is likely to find more than one solution. A solution F' is a set of features in F to build an interpreter able to execute the apps of the company. Developers can use other criteria to select one single solution, *e.g.,* the solution that can be deployed in most devices or the solution that executes faster.

4.5 Third-party Libraries

Developers often use third-party libraries to simplify their implementations and benefit from others' works. *MoMIT* handles third-party libraries because in its first step, during pre-requirement elicitation, third-party libraries are identified either by developers or through a static analysis and because, in its implementation, *MoMIT* recursively parses the source code of all JS files, including those of third-party libraries, so the JS interpreter can execute method calls from the JS app to their functions.

5 Preliminary Study

We now justify our approach through a preliminary study of the impact of a JS interpreter features on software performance metrics with the following research questions:

(PQ1) Does the selected JS-interpreter features have an impact on software performance metrics? We want to confirm an impact of each of the selected JS interpreter features and test the following null hypothesis: H_{0_1}*: there is no difference between the performance of the JS interpreter before and after modifying the default value of a selected feature.*

(PQ2) From the selected JS-interpreter features, which ones have the bigger impact on software performance metrics? We want to assess the relative impact of each feature in **PQ1** on the performance of an app and test the following null hypothesis: H_{0_2}*: there is no JS interpreter feature that has a major impact on performance metrics.*

We distinguish two categories of features: (1) ES compliant-features and (2) JS interpreter-specific features. The first category includes all ES features from version 5 to 9 as currently supported by most JS interpreters while the second is interpreter-dependent. We choose *Duktape* for its portability, compact footprint, and customization APIs, which makes it ideal for highly-constrained devices.

We consider all ES compliant-features and explore the documentation of *Duktape* to identify features related to performance. We identify 284 features; **40** of which pertain to ES compliance. Some features are binary (activated or not) while others have interval and ratio scales to tune certain features, *e.g.,* the size of the debug-code static buffer. *Exploring each feature with all their possible values is unfeasible.* Developers must use *MoMIT*.

We analyze *Duktape* configuration profiles to determine adequate starting values for features with interval and ratio scales to bootstrap *MoMIT*. These profiles provide default values for different needs: low-memory, performance-sensitive, timing-sensitive, etc. From the documentation and the profiles, we filter out features using the following criteria: deprecated features; features enabling extra debugging features; features under development and experimental features. Thus, out of 244 *Duktape* features, we consider **46** related to performance.

We develop a framework to execute a JS app and measure its performance (interpreter plus app) after (de)activating each of the 86 features one by one to study their impact on performance. We write this framework in Python, leveraging *Duktape* Python configuration script, which produces C source code and headers to compile the interpreter plus app as one C program. A first script $p1$ benchmarks the JS app using the default *Duktape* features. The second script $p2$ benchmarks the app after changing the default value of each of the 86 features individually for a value suggested either in some configuration profiles or in *Duktape* documentation to achieve a goal, *e.g.,* reduce code size. The third script $p3$ benchmarks the app using the features and values read from a configuration file. The list of features and their values and all the scripts used in this study are in the replication package [30]. We compute the size of the compiled interpreter using the Linux command `stat`, which reports the total file size, in bytes. We use the Linux *mallinfo* command to measure memory usage and `/usr/bin/time` to measure CPU time and report the total number of seconds that the process spent in user mode. Code size does not change between runs so we measure it only once. However, CPU time and memory usage may vary between runs due to CPU and operating system schedulers. Thus, we perform 10 runs to control for random errors.

The output of the scripts is a CSV file with the percentage change (δ) of each property value ($p \in P$), defined in Equation (5):

$$\delta(p) = \frac{\text{median}(p(mJSI)) - \text{median}(p(JSI))}{\text{median}(p(JSI))} \qquad (5)$$

where JSI is the JS interpreter generated using default features and $mJSI$ the miniaturized JS interpreter. Negative values indicates an improvement in p value, and positive values a detriment. We computed the median on the chosen number of runs.

We use a JS app that counts the number of prime numbers (primeSimple) below $100,000$. It is a minimal version that does not make use of strings, arrays, objects, or any libraries, but primitive types and standard arithmetic operators to be compatible with all features. The app code is in the replication package [30]. We selected a most simple app to measure the impact of each individual feature on the performance metrics, without having to handle conflicts among features: we could not measure the impact of deactivating arrays if we test an app that uses arrays. Without loss of generality, we perform the measurements in a *RPI 3* Model *B+* with a 1.4GHz 64-bit quad-core *ARM Cortex-A53* CPU, using *Duktape* 2.3.0 and *gcc 6.3.0*.

5.1 Data Analysis

Next we describe the dependent and independent variables of this preliminary study, and the statistical procedures used to address each research question.

(PQ1): Does the selected JS-interpreter features have an impact on software performance metrics? For **PQ1**, the *dependent variables* are the measured performance metrics for each JSI feature. The *independent variable* is the use of default/test values on each JSI features, which

can be dichotomy or continues values based on *Duktape* configuration files. In total we found 75 dichotomous variables and 11 continuous variables. For continuous variables, we tested two values in our experiments: the default value provided by *Duktape* JS engine and the value suggested for constrained environments (a.k.a., configuration profiles). For example, for the size of *Duktape* heap string table, DUK_USE_STRTAB_MINSIZE (see https://wiki.duktape.org/configoptions), we considered its default value (1,024 bytes) and the value in the low-memory configuration profile (128 bytes).

(PQ2): From the selected JS-interpreter features, which ones have the bigger impact on software performance metrics? For **PQ2**, we analyzed the features corresponding to the outliers obtained in **PQ1**. The *dependent* and *independent* variables are the same than in **PQ1**.

5.2 Results and Discussion of the Preliminary Study

Figure 1 shows the distribution of percentage changes in performance metrics for the 86 features, calculated using Equation (5). The numbers of features changing the percentage are, for code size, 52, memory usage, 35, and CPU time, 82. About 40% of the features impact one of the performance metrics. No feature has a percentage change equal to zero for the three performance metrics and we reject H_{0_1}.

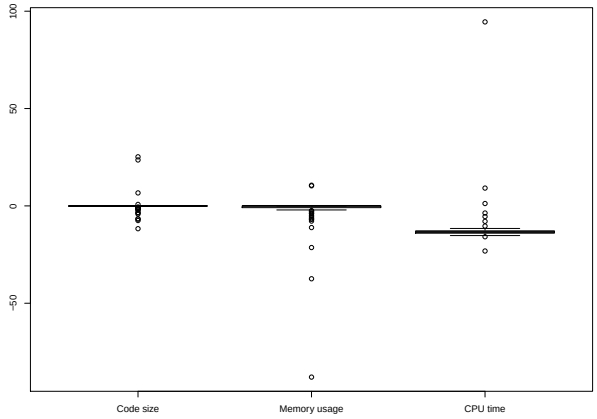

Figure 1: Percent changes for the 86 features.

Figure 2 shows the same distribution of percentage changes in the performance metrics without the outlier values. The medians of the percentage changes for *code size* and *memory usage* is small, $(-0.01\%, 0\%)$: the default values of the features have little impact on code size and memory usage for most features. The median percent change values for *CPU time* go down by -13.41%: modifying the default values of the features positively improves CPU time.

PQ1: JSI features impact performance metrics differently; all studied features impact at least one performance metric. CPU time is the performance metric most improved.

Figure 1 shows that there are many outlier features for each performance metric worthy of study in more details

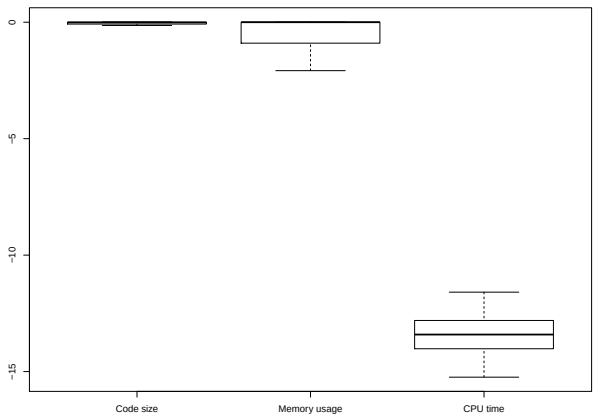

Figure 2: Percent changes for the 86 features without outliers.

in **PQ2**. We focus the discussion on features with the most extreme values. ROM objects reduce memory usage by 87.93% at the cost of 25.21% increment of file size. The maximum improvement in code size is 11.71% with Feature 3, DUK_USE_EXEC_PREFER_SIZE. The maximum increase of memory usage is 10.62% with reference counting (garbage collection). Regarding performance, the maximum improvement is reached with Feature 84, DUK_USE_FASTINT, with 24% while the worse is 94.5% due to Feature 6, DUK_USE_PREFER_SIZE. Thus, we reject H_{0_2}.

PQ2: We identified 90 features impacting the performance metrics by more than 5%. Table 1 summarises the most impacting metrics.

Table 1 presents the two features most impacting CPU time. The complete list of features and performance metrics is in our replication package [30]. In Table 1, *ID* is an arbitrary ID assigned to each feature for convenience; *Name* is the feature name; *Description* is a summary of the feature based on the documentation; *Value* is the tested value in our experiments (different from *Duktape* default value); and the last three columns show the percentages changes of the performance metrics.

Feature 3 reduces code size $\approx 12\%$ by increasing executing time around 9%; Feature 11 reduces code size 7% and CPU time 12% while increasing memory usage $\approx 1\%$. These are the features that reduce the most code size. Using ROM built-ins (Features 7–10) increases code size 25% and Feature 86 $\approx 24\%$. The use of ROM built-ins reduce memory usage by about 88%, the highest reduction for all features and performance metrics. Features 5, 43, and 19 reduce memory usage by 37%, 21%, and 11%, respectively. Feature 11 increases memory usage by 11%, the only feature increasing memory usage by more than 5%. It is related to the use of automatic reference counting for garbage collection. Features 15 and 84 provide the highest reduction in CPU time, with 23% and 16%, respectively. Feature 6 reports the worst results of all features, increasing CPU time by 94%. Practitioners must avoid to activate DUK_USE_PREFER_SIZE. We observe that

different combinations of features impact differently the performance of a JS app. The (de)activation of some features results in a high improvement of a metric while worsening one or more metrics. Only Feature 5 presents no conflict with any other metric but reduces compliance.

The preliminary study is useful for practitioners to make informed decisions about the features to (de)activate from their JSI when porting JS apps to IoT devices.

Conclusion: there is a conflict between the three performance metrics studied that shows the need to provide practitioners with an automated approach to select the features that are more advantageous for the devices that they are targeting.

6 IMPLEMENTATION

We describe the techniques and tools used for implementing *MoMIT* to miniaturize JS interpreters for IoT constrained devices. Without loss of generality, we focus on *Duktape* as the JS engine to generate the miniaturized JS interpreters.

6.1 Pre-requirement Elicitation

As described in Section 4, a miniaturized JS interpreter is comprised of compulsory and optional features. The list of compulsory features can be inferred through a manual inspection of the JS code and–or by asking the authors to provide pre-requirements, or by using a JS parser, and–or feature detection tools. The optional features form the search space of the problem and may differ between JS interpreters.

6.2 Selection of IoT Device Candidates

We used the devices already introduced in Table 2 as the IoT device candidates used in this work. We rank them to set the preference that *MoMIT* will use to filter solutions. In Table 3, we arbitrarily set the preferences of devices based on price, size, cloud connections, and company support of the candidate devices, from which *Photon* is the more attractive, while BeagleBone Black is the less attractive due to its price, size, and lack of cloud support. We did not consider the processing power or memory capacity, as we assume that if a JS app can be miniaturized for the SBCs, it will be fit as well on the SoCs devices, so does not make sense to prioritize the miniaturization to fit them.

Table 3: IoT devices used in this work

Device Name	Preference
Photon	5
ESP32	4
JN5168	3
RPI 3 ModelB+	2
BeagleBone Black	1

6.3 Feature Identification

Another contribution of this work is the classification of *Duktape* interpreter options in four categories based on their impact on performance and compliance with ES standards. This classification is valid for any JS interpreter.

6.3.1 ECMAScript Compliance Options

Features in this category provide built-in functions to comply with ES standards. Enabling/Disabling some of these features impacts the level of compliance of the JS interpreter. An interpreter that does not comply with the ES for which the app was written may fail or have unexpected behaviour. For example, ES6 requires to accept *HTML* commenting style and this feature is not available on ES5.

6.3.2 Code Size Options

Features in this category reduce code size of the JS interpreter at the cost of some of its features, *e.g.,* disabling Unicode support for non-BMP characters reduces code size.

6.3.3 Memory Performance Options

Features in this category reduce the memory footprint of the JS interpreter at the cost of CPU speed and–or code size. For example, compiling objects and–or strings as constants and storing them in ROM reduces startup RAM usage considerably at the cost of code size and slower performance.

6.3.4 CPU Performance Options

Features in this category increase speed by reducing CPU time at the cost of memory and–or code size, *e.g.,* a look-up table to convert objects-to-strings using `JSON.stringify` improves performance but takes storage space.

6.4 Selection of Feature Combinations

In the last step of *MoMIT*, we describe solutions as bit-vectors $\vec{x} = \{x_1, \ldots, x_{|OF|} \in 0, 1\}$, where x_j indicates if feature $f_j \in OF$ is included in the solution, with $\vec{x} : x_j = 1$ if it is, else 0. Let $\vec{x}$ be a candidate solution and Sel a function mapping a bit-vector $\vec{x}$ into the set of features F'. We define CDR (customer's dissatisfaction rate, $-CSR$), CS (code size), MU (memory usage), and PT (CPU time) as:

$$\begin{aligned} CDR(\vec{x}) &= -CSR(Sel(\vec{x}) \cup ComF) \\ CS(\vec{x}) &= CS(Sel(\vec{x}) \cup ComF) \\ MU(\vec{x}) &= MU(Sel(\vec{x}) \cup ComF) \\ PT(\vec{x}) &= PT(Sel(\vec{x}) \cup ComF) \end{aligned}$$

where $CS(\vec{x})$, $MU(\vec{x})$ and $PT(\vec{x})$ are the code size of the JS interpreter, the memory usage, and the CPU time, respectively, of the JS app with the features in $\vec{x}$ and the compulsory features $ComF$. The direction of the optimization for the objectives is to minimize their values, thus we we define $CDR = CSR \times -1$. A solution is a vector $\vec{x}$ whose elements are Pareto optimal.

6.5 Dependencies among JS Interpreter Features

One of the contributions of this work is the identification of 10 hidden dependencies among the selected features, presented in Table 4 with their IDs and brief descriptions. These dependencies, if ignored, prevent the generation of valid JS interpreters. We identified them through our preliminary study, in which we evaluated each feature independently, and from our evaluation of *MoMIT*, during which we ran

the search process with the complete list of features and detected compilation/execution errors. These dependencies are not exhaustive but include the minimum sets to consider when evaluating *MoMIT* using 23 JS tests.

We found that the most dependent (and restrictive) features are related to ROM built-ins (IDs 7-10). When activated, these features modify the values of other features, which results in broken JS interpreters. These features were commented out by the authors of *Duktape* in the low-memory environments profile to avoid errors. These features appear also in a profile that only activates these features. While some undocumented dependencies among features seem related due to their name (*e.g.,* JSON built-in depends on JSON support, IDs 26 and 27), other dependencies are not obvious (*e.g., Global* built-in depends on the *Number* built-in, IDs 24 and 30).

Table 4: Dependencies among features in *Duktape*.

ID	Description
26, 27	Disabling JSON built-in requires to disable JSON support too, but not the other way around
32, 34	Disabling regular expression support requires to disable the string built-in as well
7–10	Storing String, and Objects in ROM requires to enable ROM Global inherit and disabling Hstring Array index feature
11, 14	Disabling reference counting (garbage collection) requires to disable the use of double linked heap.
72–74	Minimum, Maximum and shrink limit for duktape heap string table have to be set to the same value to avoid resizing during runtime
16, 20	These options provide support for augmenting ES error objects to comply with ES 5, and have to be deactivated together
17, 21	These options augment an ES error object at throw time and have to be deactivated together
31, 24	Disabling duktape object built-in requires to disable global built-in too
24, 30	Disabling global built-in requires to disable number built-in too

To avoid evaluating solutions unfeasible due to dependencies among features, we devise a mechanism to repair solutions when evaluating candidates. This mechanism detects the default value of any of the features with dependencies has changed: it did, it adjusts the feature value according to Table 4. For example, if it detects that Feature 26 has been deactivated, then it deactivates Feature 27 as well. It treats differently Features 7–10 and 72–74: if ROM built-ins are activated, then the rest of the features must keep their default values. We presume that this observation is the reason for the authors of *Duktape* to provide a separate configuration file with only these features activated (*roms-builtins.yaml*). If Features 72–74 change, which define the maximum, minimum and shrink limit of *Duktape* heap string table, then it resets them all to their minimum values as suggested in the *low_memory.yaml* profile of *Duktape*.

7 Evaluation of *MoMIT*

We now evaluate the effectiveness of *MoMIT* at miniaturizing JS interpreters to run on IoT devices based on the code features requirements of JS apps. The *quality focus* is the reduction of code size, memory usage, and CPU time to fit a JS app in some IoT devices while considering customers' preferences. The *perspective* is that of companies and individuals interested in porting their exiting JS apps to IoT devices. The *context* consists of 23 JS tests belonging to a JS benchmark (SunSpider 1.0.2) to test the core JS language [31]. We selected SunSpider because: (1) it is a complete benchmark to test the JS language; (2) it is a balanced, real-world test suite focusing on real developers' problems, including math computation, string processing, JSON parsing, etc.; (3) it does not make any assumptions about the architecture of the system; and, (4) it has been used to compare the performance of different Web browsers.

Table 5 presents the tests in our evaluation, with their compulsory features. It also shows baseline memory usages and CPU times when using default *Duktape* features. The last column shows the number of devices on which the JS tests can be ported based on the values in Table 3. We omit code size because it is the same for all JS tests when using default features: **570 KB**. We excluded three tests because they include files of more than 100 KB each and, thus, could not fit within the space limitations of *JN5168* and its 32 KB, making it impossible for *MoMIT* to find any solution.

Table 5: 23 JS tests from SunSpider 1.0.2.

JS test	$ComF$	MU (KB)	PT (Sec.)	Devices
3d-cube	15; 29; 34;31	166.496	0.205	3
3d-morph	15; 29	132.000	0.460	3
3d-raytrace	15; 19; 29;31	387.936	0.250	3
access-binary-trees	29	179.440	0.235	3
access-fannkuch	15;31;34	132.704	0.470	3
access-nbody	15; 29	146.848	0.415	3
access-nsieve	15	131.296	0.140	3
bitops-3bit-bits-in-byte		128.176	0.440	3
bitops-bits-in-byte		128.528	0.465	3
bitops-bitwise-and		126.144	1.275	4
bitops-nsieve-bits	15	130.624	0.775	3
controlflow-recursive		184.832	0.210	3
crypto-aes	15; 19; 29; 34; 32	178.216	0.210	3
crypto-md5	15; 34;32	176.544	0.355	3
crypto-sha1	15; 34;32	163.088	0.340	3
date-format-tofte	15; 19;24;31;34;32;7;8;9;10	1,817.200	0.535	2
date-format-xparb	19;24;31;34;32;7;8;9;10	183.824	0.335	3
math-cordic	19	132.720	0.475	3
math-partial-sums	29	129.968	0.495	3
math-spectral-norm	29	144.240	0.190	3
string-base64	29; 34;32	275.632	0.725	3
string-fasta	15; 34;32	140.064	0.875	3
string-validate-input	15; 29; 34;32	614.464	1.160	2
Median				**3**

We use two implementations of *MoMIT*, one with the NSGA-II algorithm and a dedicated repair function (*cf.,* Section 6.5), the other with a random search with repair, which we name *RS+*. We chose NSGA-II because it was successfully applied in many previous works. We use *RS+* because it is a simple approach, easy to implement, that serves as baseline for NSGA-II. We included a repair function in *RS+* to ensure that *MoMIT* produces valid solutions (the odds of generating valid solutions using a pure random search is minimal, as discussed in previous works [10], [11]).

7.1 Research Questions

We ask the following research questions in our evaluation:

(RQ1) To what extent can *MoMIT* miniaturize JS tests to run on constrained devices? We want to quantify the improvement of the performance metrics after miniaturizing the JS interpreter and, based on these measures, to determine the number of IoT devices on which the JS

apps can run. We compute the percentage changes of the performance metrics before and after miniaturization with Equation 5 for CS, MU, and PT metrics. We define NDA as the difference between numbers of devices running the JS app before and after miniaturizing the JS interpreter. We consider a JS app to be ported to a device d if its code size and memory usage are less than or equal to the storage and memory capacity of d for one or more solutions.

(RQ2) What is the most convenient algorithm to instantiate *MoMIT*? We want to identify the best algorithm to instantiate *MoMIT* in terms of CPU time and quality of the solutions. We compute the hypervolume (HV) [32] and Pareto front size (PFS) indicators to measure the quality of the solutions. HV considers the convergence and diversity of the resulting approximation set. Higher values of HV are desirable. PFS measures the number of solutions included in a Pareto front. Higher PFS indicates that an algorithm finds more non-dominated solutions. We compute HV and PFS over the Pareto reference front obtained for each JS test, *i.e.,* the non-dominated solutions found over several runs by all the algorithms evaluated on a given JS test. We compute the non-parametric Mann–Whitney U test at 5% significance level of confidence to determine the significance of the obtained results. We correct the obtained p-values with the Bonferroni p-value adjustment procedure [33] to reduce the risk of Type-I error (corrected $\alpha = 0.017$).

(RQ3) Can *MoMIT* miniaturize apps involving third-party libraries? Real world JS apps include third-party libraries to reuse existing functionality and provide more unique features. We study whether *MoMIT* can miniaturize JS apps involving third-party libraries using the same metrics than RQ1. We extend one of the JS tests in RQs 1 and 2 (*date-format-tofte*) to format and manipulate numbers using *Numeral.js [34]* from node.js. We choose node.js because it is non-trivial, with more than seven years of development. It addresses common problems for JS developers. We name our extension of *date-format-tofte*, *date-format-tofte-plus*.

7.2 Tuning of Parameters

As with any search algorithm, we must tune the control parameters of the algorithms used by *MoMIT*. We must keep the numbers of evaluations and of individuals (for NSGA-II) relatively low to produce solutions in reasonable amounts of time. Indeed, *MoMIT* must build a JS interpreter using *Duktape* Python script, which takes about 20 seconds on our test machine, then compile the harness code, and execute the app 10 times to average the memory and CPU usages (which may vary between executions).

The transformation operators used for NSGA-II are single-point crossover and bit-flip mutation; the selection operator is binary tournament [25]. We ran a grid search [35] for the three parameters: population size (μ), crossover probability (CXPB), and mutation probability (MUTPB). Our grid search space considered the following values: $\{\mu$ = [10]$\}$ $\times$ $\{$CPBX = [0.6, 0.7, 0.8, 0.9]$\}$ $\times$ $\{$MUTPB = [0.1, 0.15, 0.2, 0.25]$\}$. We use HV as quality indicator. We found the highest HV with the following parameters: μ = 10, CPBX = 0.8, and MUTPB = 0.1.

Also, we run *MoMIT* 30 times on each app to reduce the observational error. We report median values for the grid search and in the evaluation of *MoMIT*. We use the number of fitness-function evaluations as stopping criteria. As the number of evaluations increase, *MoMIT* obtains better quality solutions on average. This increase in quality is usually very fast when the maximum number of evaluation is low because the slope of the curve showing quality versus maximum number of evaluations is high at the beginning of the search. The slope then tends to decrease as the search progresses. We set the number of evaluations to 250 because, with this value, the slope of the curve is *low enough*, *i.e., MoMIT* cannot miniaturize more a JS interpreter to fit the most constrained device.

We implemented NSGA-II and *RS+* with JMetalPy, a framework for multiobjective optimization in Python [36].

8 Results

We now answer the research questions.

8.1 RQ1: To what extent can *MoMIT* miniaturize JS tests to run on constrained devices?

Table 6 shows the results obtained from *MoMIT* after miniaturizing the 23 JS tests. The values presented in Columns 2–4 are median values and the corresponding interquartile ranges (IQRs) obtained from the Pareto reference front. Column 5 shows the number of new IoT devices on which the JS tests can run after miniaturization. Column 6 shows the total number of devices on which the JS tests run.

We observe that the code sizes of 20 out of the 23 tests were reduced by more than 20% with *math-partial-sums* the test with the maximum reduction (36.58%). Memory usage exhibits the highest reduction of the three metrics with a median of 55.51%, and a maximum reduction of 92.93% for *date-format-tofte*, which has the largest number of compulsory features and, thus, the smaller search space for *MoMIT*. CPU time reduction has a median of 35.71% with a maximum reduction of 76.34% for *bitops-bits-in-byte*.

The median number of IoT devices on which we can deploy the JS tests is three, using *Duktape* default configuration values (*cf.,* Table 5). It reaches four out of five devices after miniaturization. The JS tests for which NDA increased by more than one device are *date-format-tofte* and *string-validate-input*. For these two tests, the memory usage was higher than the capacity of all of the IoT devices before miniaturization. *MoMIT* generated interpreter allowing to run these JS apps to most devices. For *string-validate-input*, five solutions (out of 66) on the Pareto front improve its memory usage by more than 70%.

Three tests could not fit in more IoT devices despite the reductions because *MoMIT* could not reduce sufficiently the footprint of the JS interpreter: *3d-raytrace*, *bitops-bitwise-and*, and *string-base64*. For the first and third tests, the memory improvement was not enough to fit within the memory of the *Photon*. For the second, the original test already fits on four of the five devices and running it on the all devices would require reducing by more than half the original code size and using one third of the memory.

MoMIT could not fit any JS test to the *JN5168* microcontroller due to its low memory and storage capacities.

RQ1: MoMIT *can improve performance metrics of JS interpreters by removing unnecessary features to execute JS apps in more constrained devices without modifying their original source code.*

8.2 RQ2: What is the most convenient algorithm to instantiate *MoMIT*?

Table 7 shows the computation times of the two compared algorithm, without p and δ values because **all the differences were statistically significant with large effect size.**

NSGA-II performs faster than *RS+* because the generic NSGA-II operators generate more repeated solutions in comparison to *RS+* and, thus, explore less solutions. For both algorithms, we store in a dictionary the objective values of each evaluated solution to reduce computation time: when an algorithm finds an existing solution, it retrieves the objective values stored in the dictionary directly. The search algorithms *per se* are not the limiting factors in *MoMIT* but the compilation of the JSI and execution of the JS apps.

Table 8 shows the numbers of non-dominated solutions contributed by each algorithm to the Pareto reference front. The numbers of solutions contributed by *RS+* overcome those of NSGA-II, indicating poor performance of the latter one, according to PFS. Finally, Table 9 reports the average hypervolume values for each algorithm. It shows that *RS+* overcomes NSGA-II.

We also compared NDA for NSGA-II and *RS+* and report that *RS+* overcame NSGA-II for two tests only: *date-format-tofte* and *string-validate-input*.

Results show that *RS+* overcomes NSGA-II, which is a well-known and widely used evolutionary algorithm, possibly due to the limited numbers of evaluations and the small population size that could prevent NSGA-II to find better solutions. To identify the reasons for the low performance of NSGA-II and considering the high cost of more evaluations, we ran a microexperiment using only two randomly-selected JS tests and applied both NSGA-II and *RS+*. We included an additional search algorithm, SWAY [11], designed for situations where evaluating solutions in the search space is expensive but with a high correlation between decision and search spaces.

SWAY performs most of its evaluations in the decision space and limits the number of objective evaluations compared to EAs. We used the original implementation of SWAY[7]. We recast the problem of miniaturization as a software product line (SPL) in which each combination of features forms a product, *i.e.,* a JS interpreter. We defined a list of Boolean predicates to model the dependency among JS interpreter features and compulsory features for each selected JS test using the conjunctive normal form (CNF). The predicates defined a valid solution. We used the split function designed for binary decision spaces and inspired by research on radial basis function kernel [37].We then fed the Boolean predicates to a SAT solver responsible for finding all satisfiable CNF expressions within the size of the population. Then, SWAY clustered the population according to their decision variables using a radial coordinate system and obtained solutions [11].

7. https://github.com/ginfung/FSSE

For all three algorithms, we performed 10 runs with 250 and 15,000 evaluations. We set a population of 100 for NSGA-II. We use same control parameters for SWAY as defined by its authors (except for number of evaluations).

Table 10 shows the execution times (ETs) and quality metrics of the solutions from the microexperiment. We recomputed HV with only 3 objectives. With respect to ETs, SWAY is the fastest approach, with only 8 and 11 minutes (median values), because it performs less objective evaluations compared to the other algorithms. NSGA-II is the second fastest, with execution times of about 27 and 42 hours. *RS+*is slowest with 67 and 105 hours approximately. Results for HV are similar for 250 and 15,000 evaluations: *RS+* outperforms NSGA-II, which outperforms SWAY. *RS+* reports higher gain for PFS for non-dominated solutions added to the Pareto front; NSGA-II contributed few dominated solutions and SWAY none.

The HV metric for NSGA-II considerably improved from 0.49 to 0.82 for *access-fannkuch* JS test. It remained the same for *3d-cube* due to its mandatory features. We thus conclude that increasing the number of evaluations for NSGA-II and *RS+* increases HV and PFS values at the costs of time. The number of new devices reached after miniaturizing remained the same (Column 5) than with 250 evaluations. SWAY performed faster but could not reach the same number of devices for *3d-cube*. It did not contribute any new solutions to the Pareto front: the solutions generated by SWAY were all dominated by those of the other algorithms.

After considerably augmenting the number of evaluations and obtaining similar results, we suggest that the inferior performance of NSGA-II in comparison with *RS+* is caused by the transformation operators (crossover and mutation) employed, which are typically used for binary solutions, but not necessary the most suitable ones for this particular problem. With respect to SWAY, we are surprised by the low quality of the obtained results. We suggest this is the result of assuming that there is a direct relationship between the decision variables and the objective values, *i.e.*, the fact that SWAY clusters solutions based on the numbers of ones/zeros might be appropriate for SPLs where "one" means adding a feature/component and "zero" otherwise. On the other hand, in the context of *Duktape* features the concept of "one" equal to adding extra functionality does not hold. For example, consider feature 5 from Table 1, which default value is "zero", and when switch it to "one" removes ES compliance to reduce memory usage. In other words, having more "ones" does not imply adding more functionality to the interpreter in all cases.

RQ2: The best algorithm for MoMIT *is* RS+ *by the quality of its solutions and the number of devices enabled. NSGA-II performs faster than* RS+ *and can reach almost the same number of devices. SWAY is the fastest but worst in terms of quality metrics.*

8.3 RQ3: Can *MoMIT* miniaturize apps involving third-party libraries?

Table 11 presents the results of applying *MoMIT* to *date-format-tofte-plus*, which uses *Numeral* of the node.js library. *MoMIT* can reduce code size, memory usage, and CPU

Table 6: Results of the miniaturization process using *MoMIT* on 23 JS tests

JS Test	δCS (%)		δMU (%)		δPT (%)		NDA	Devices
3d-cube	-5.67	(54.37)	-55.51	(15.56)	-31.71	(14.02)	1	4
3d-morph	-12.56	(59.18)	-67.26	(19.96)	-34.78	(8.69)	1	4
3d-raytrace	-28.19	(16.88)	-15.89	(293.95)	-8.00	(24.00)	0	3
access-binary-trees	-7.64	(60.37)	-49.63	(15.24)	-31.91	(2.13)	1	4
access-fannkuch	-30.13	(13.43)	-53.94	(35.17)	-36.70	(27.92)	1	4
access-nbody	-30.67	(12.12)	-50.93	(17.73)	-37.35	(14.46)	1	4
access-nsieve	-30.94	(23.61)	-68.45	(6.34)	-35.71	(28.57)	1	4
bitops-3bit-bits-in-byte	24.72	(50.28)	-82.61	(6.16)	-27.27	(0.00)	1	4
bitops-bits-in-byte	-33.62	(14.28)	-69.53	(7.48)	-76.34	(16.13)	1	4
bitops-bitwise-and	-34.57	(41.73)	-75.94	(5.44)	-34.12	(14.70)	0	4
bitops-nsieve-bits	-31.21	(50.68)	-66.62	(72.32)	-34.19	(59.03)	1	4
controlflow-recursive	-33.89	(12.27)	-63.58	(12.34)	-19.05	(19.05)	1	4
crypto-aes	-28.97	(11.30)	4.56	(48.39)	-26.60	(51.06)	1	4
crypto-md5	-31.59	(21.24)	-35.54	(59.06)	-60.56	(28.17)	1	4
crypto-sha1	-24.58	(56.96)	-51.31	(21.88)	-41.18	(25.00)	1	4
date-format-tofte	-31.03	(4.18)	-92.93	(3.36)	-55.14	(20.09)	2	4
date-format-xparb	-30.59	(13.57)	-25.78	(33.56)	-37.31	(11.57)	1	4
math-cordic	-35.57	(11.68)	-61.37	(12.99)	-43.16	(24.74)	1	4
math-partial-sums	-36.58	(10.22)	-62.58	(7.57)	-49.49	(6.82)	1	4
math-spectral-norm	-9.06	(58.78)	-63.18	(15.72)	-42.11	(5.26)	1	4
string-base64	-32.1	(7.29)	-22.8	(892.29)	-24.14	(15.52)	0	3
string-fasta	-28.62	(6.19)	-50.11	(37.20)	-37.14	(16.57)	1	4
string-validate-input	-29.53	(6.93)	-11.34	(6.82)	-12.43	(21.16)	2	4
Total Median (IQR)	**-30.59**	**(38.99)**	**-55.51**	**(28.66)**	**-35.71**	**(12.07)**	**1**	**4**

Table 7: Execution times of the search algorithms in secs. (All differences statistically significant with large effect sizes.)

JS Test	NSGA-II	*RS+*
3d-cube	2,642	5,035
3d-morph	2,987	5,948
3d-raytrace	2,679	4,253
access-binary-trees	2,419	5,004
access-fannkuch	3,198	6,511
access-nbody	2,606	5,375
access-nsieve	3,163	6,272
bitops-3bit-bits-in-byte	2,547	4,800
bitops-bits-in-byte	2,571	4,982
bitops-bitwise-and	3,462	6,843
bitops-nsieve-bits	2,834	5,619
controlflow-recursive	2,340	4,778
crypto-aes	2,836	4,702
crypto-md5	2,786	3,978
crypto-sha1	2,785	4,009
date-format-tofte	2,461	4,375
date-format-xparb	2,536	3,974
math-cordic	2,857	4,261
math-partial-sums	2,879	4,122
math-spectral-norm	2,534	3,736
string-base64	3,947	6,328
string-fasta	3,285	4,833
string-validate-input	9,793	16,791
Total Median	**2,786**	**4,833**

Table 8: Pareto optimal solutions by NSGA-II and *RS+*.

JS test	Solutions	NSGA-II		*RS+*	
3d-cube	102	39	(38.24%)	63	(61.76%)
3d-morph	73	35	(47.95%)	38	(52.05%)
3d-raytrace	81	3	(3.70%)	78	(96.30%)
access-binary-trees	89	47	(52.81%)	42	(47.19%)
access-fannkuch	66	7	(10.61%)	59	(89.39%)
access-nbody	44	13	(29.55%)	31	(70.45%)
access-nsieve	33	8	(24.24%)	25	(75.76%)
bitops-3bit-bits-in-byte	71	29	(40.85%)	42	(59.15%)
bitops-bits-in-byte	40	10	(25.00%)	30	(75.00%)
bitops-bitwise-and	36	9	(25.00%)	27	(75.00%)
bitops-nsieve-bits	52	7	(13.46%)	45	(86.54%)
controlflow-recursive	40	7	(17.50%)	33	(82.50%)
crypto-aes	58	6	(10.34%)	52	(89.66%)
crypto-md5	37	3	(8.11%)	34	(91.89%)
crypto-sha1	63	6	(9.52%)	57	(90.48%)
date-format-tofte	24	2	(8.33%)	22	(91.67%)
date-format-xparb	66	4	(6.06%)	62	(93.94%)
math-cordic	55	9	(16.36%)	46	(83.64%)
math-partial-sums	36	1	(2.78%)	35	(97.22%)
math-spectral-norm	53	1	(1.89%)	52	(98.11%)
string-base64	55	11	(20.00%)	44	(80.00%)
string-fasta	37	1	(2.70%)	36	(97.30%)
string-validate-input	66	11	(16.67%)	55	(83.33%)

time by 22.89%, 77.41%, 13.95%, respectively. It can port the extended JS test to one more device than the original test.

MoMIT cannot port it to more devices because the JS app now include the third-party library, which brings overhead in terms of code size and memory usage. Thus, we show that *MoMIT* works as well with third-party libraries but that these libraries limit its search space and constrain the number of devices on which it can port JS apps. We discuss further third-party libraries in Section 9.

The process of miniaturizing JS apps that make use of third-party libraries is an iterative process during which developers must identify JS engine features required to run their app with the libraries that they are importing. They must implement *Duktape* function `duk_ret_t mod_search(duk_context *ctx)`[8] so that it finds the third-party libraries. This function allows *Duktape* to parse the third-party libraries and register them as global objects to use their functionalities in the JS app.

Adding third-party libraries and, thus possibly, additional features could add overhead in terms of code size, memory usage, and CPU time when compared to an app not requiring these libraries. However, this also reduces the search space of the problem. In our study, we mitigated the impact incurred by loading Numeral.js in memory by minifying[9] the file, which reduced it from 33Kb to 12Kb.

8. https://github.com/svaarala/duktape/blob/master/extras/module-duktape/README.rst

9. Minifying is a typical practice in JS, during which developers remove spaces and indentation to reduce the size of a JS file to improve performance.

Table 9: Average HV values, significance and effect size.

JS Test	NSGA-II	*RS+*	p-value	δ
3d-cube	0.78	0.88	2.31E-10	Large
3d-morph	0.85	0.90	5.34E-08	Large
3d-raytrace	0.56	0.83	1.69E-17	Large
access-binary-trees	0.74	0.80	3.03E-03	Medium
access-fannkuch	0.49	0.70	2.49E-13	Large
access-nbody	0.27	0.37	8.00E-06	Large
access-nsieve	0.33	0.46	3.02E-05	Large
bitops-3bit-bits-in-byte	0.19	0.38	3.43E-08	Large
bitops-bits-in-byte	0.19	0.34	1.09E-06	Large
bitops-bitwise-and	0.28	0.44	5.18E-09	Large
bitops-nsieve-bits	0.45	0.61	1.09E-06	Large
controlflow-recursive	0.17	0.32	7.24E-10	Large
crypto-aes	0.77	0.91	1.69E-17	Large
crypto-md5	0.42	0.74	1.69E-17	Large
crypto-sha1	0.43	0.72	2.03E-16	Large
date-format-tofte	0.34	0.78	1.69E-17	Large
date-format-xparb	0.34	0.80	1.69E-17	Large
math-cordic	0.23	0.44	9.75E-14	Large
math-partial-sums	0.21	0.46	1.69E-17	Large
math-spectral-norm	0.17	0.38	3.21E-16	Large
string-base64	0.87	0.92	1.02E-07	Large
string-fasta	0.37	0.84	1.69E-17	Large
string-validate-input	0.84	0.92	1.09E-09	Large

We identified five JS engine features required to use Numeral.js library. These features are related to JSON and math functionalities taken from *Duktape* builtins, which were not required before adding Numeral.js to date-format-tofte. All these features are disabled on *Duktape* low memory profile configurations, which means that adding Numeral.js mostly impacted memory usage. Yet date-format-tofte-plus still run on the same number of devices than the original test date-format-tofte. MoMIT could reduce 93% of memory usage, which translates into one additional device (ESP23) to deploy it, compared to using *Duktape* default features.

RQ3: MoMIT *can miniaturize JS apps that use third-party libraries.*

9 Discussion

We now discuss the results of evaluating *MoMIT*.

9.1 *MoMIT* Extensibility

CPython, a C implementation of Python, was manually customized to generate Python interpreters for embedded devices (*e.g.*, *PyMite* and *TinyPython*) by removing unessential features and supporting a subset of Python syntax. *MoMIT* can be extended to CPython, or other programming languages using interpreters/virtual machines, by adapting it to generate, *e.g.*,, CPython interpreters and providing the PRs of Python apps. Thus, developers do not need to miniaturize manually CPython and can focus on their apps.

Miniaturization in general is about reducing storage, memory, and CPU usages. Previous approaches, *e.g.*, MoMS [18], focused on apps without considering their runtime support. They work well for apps compiled directly into executables but are not adequate for interpreters/virtual machines, for which *MoMIT* works well. Hence, *MoMIT* is complementary to previous works.

Listing 1: Compiling *harness* using GCC with default options options

```
gcc −std=c99 −o harness harness.c duktape.c −lm
```

Listing 2: Compiling *harness* using GCC with optimized flags for reducing code size on ARM devices

```
gcc −o harness −m32 −std=c99 −Wall −Os −fomit−frame−pointer
    −flto −fno−asynchronous−unwind−tables −ffunction−sections
    −Wl,−−gc−sections −fno−stack−protector −Iduktape−src
    duktape.c harness.c −lm
```

Compilation options also complement *MoMIT* and previous works. We used default compilation options to be conservative and leave their tuning to developers. Hence, *MoMIT* is general without preventing further optimisations. For example, we compiled the *harness* using default GCC options (as in our evaluation, *cf.*, Listing 1 and with options optimized for ARM processors (*cf.*, Listing 2) and observed that code size went down from 362 KB to 132 KB.

9.2 *MoMIT* Caveats and Limitations

We observed that some programming practices require certain JS features, which reduce performance. For example, *date-format-tofte* modifies the *Date prototype*, which prevents using ROM built-ins because they make the Date prototype unmodifiable. Rewriting the app to remove such practices would allow *MoMIT* to minimize further the JS interpreter.

We showed that *MoMIT* can handle apps using third-party libraries. However, they must be used with caution: if the use of a third-party library is small, developers could reimplement its functionalities directly in C in the JS interpreter to reduce code size and memory usage.

Some IoT devices have been steadily promoted to full-fledged computers thank to advances in computing hardware and battery technologies. The *RPI* is used in many hobby, industrial, and research projects. Hence, developers should consider their capabilities even if *RPI*s and other such computers can hardly be used in many scenarios because of their form factors, their energy consumptions, their costs, and other considerations. For example, it is not environmentally-friendly and cost-effective to drop hundreds or thousands of *RPI*s in forests to monitor droughts and fires when compared to *Photon* or *ESP32*. *RPI*s should remain at the "edge" of an IoT systems.

10 Threats to validity

We following common guidelines for empirical studies [38].

Construct validity concerns the relation between theory and observations. Imprecision in the measurements performed in the evaluation could threaten this validity. We measured code size, memory usage and CPU time using well-known Linux commands and repeated the measurements 10 times for each JS app execution and 30 times when applying *MoMIT* in Section 7. We cannot exclude the impact of the operating system, which we mitigated by performing multiple executions in a dedicated *RPI*, disconnected from the Internet, and running only the tools and scripts used for the evaluation.

Table 10: Execution times and quality metrics of the microexperiment (median ETs and mean HV).

JS Test	EA	250 evaluations			15,000 evaluations			NDA
		ET (sec.)	HV	PFS	ET (secs.)	HV	PFS	
3d-cube	NSGA-II	2,642	0.67	4	96,321	0.67	7	1
	RS+	5,035	0.90	55	240,132	0.94	107	1
	SWAY	N/A	N/A	NA	488	0.49	0	0
access-fannkuch	NSGA-II	3,198	0.49	2	149,514	0.82	15	1
	RS+	6,511	0.73	51	377,487	0.85	69	1
	SWAY	N/A	N/A	N/A	680	0.24	0	1

Table 11: Median quality metrics of *MoMIT* applied to a JS test that uses a third-party library.

JS Test	δCS	δMU	δPT	NDA	Devices
date-format-tofte-plus	-22.89%	-77.41%	-13.95%	1	3

Internal validity pertains to the chosen JS apps, used tools, and applied analysis methods. We used a particular yet representative subset of JS tests as proxy for JS apps. We used well-known theory and measurements to ensure statistical validity of the performance metrics. We used NSGA-II to perform our benchmarks. However, multi-objective optimization algorithms exist with possibly better performance, like MOEA/D or MOEA/D-DE [39]. Hence, we may not have achieve the best performance and future works includes using these recent algorithms.

Conclusion validity concerns the relation between the treatment and the outcome. We tested and met the assumptions of the constructed statistical models: *e.g.*, non-parametric tests, Mann-Whitney U Test and Cliff's δ, that do not make assumptions on the underlying data distributions.

Reliability validity threats concern the possibility of replicating this study. The tools used in this study are open-source and can be accessed with the data collected an generated in the online replication package [30] .

External validity regards the generalization of our results, which must be interpreted carefully and could depend on the specific devices, operating systems, and JS apps. For example, if an app needs specialised hardware, such as a GPS, then *MoMIT* has a reduced search space to miniaturize this app because of this hardware constraint. Although, we miniaturized JS interpreters, *MoMIT* applies to other interpreters, *e.g.*, CPython. We discussed all the results and put them in perspectives.

11 Conclusion and Future Work

We presented *MoMIT*, an automated, multiobjective approach for miniaturizing JS apps to run on constrained IoT devices. *MoMIT* supports companies wanting to deploy their apps on different devices from the same source code.

First, we performed a preliminary study to identify the features impacting the performance in terms of storage, memory usage, and CPU time of JS apps running on the *Duktape* interpreter. We identified **86** features out of 283. We identified 10 hidden dependencies among 20 of the 86 features and corrected a bug[2] preventing the compilation of *Duktape* in some scenarios. Then, we formulated miniaturization as a multiobjective problem, implemented using NSGA-II, *RS*+, and SWAY.

We evaluated *MoMIT* by miniaturizing 23 JS tests from SunSpider and showed that it could reduce code size, memory usage, and CPU time by 31%, 56%, and 36%, respectively (medians). ***MoMIT* could miniaturize 21 JS test apps out of 23 to run on SoCs the size of a quarter coin: *Photon* and *ESP32*.** We extended one of these apps to use the popular node.js third-party library and showed that *MoMIT* could also miniaturize apps using such libraries.

We released the source code of *MoMIT* as open-source so researchers and practitioners can benefit from our work and replicate our evaluation [30].

We want to expand *MoMIT* to other interpreters, *e.g.*, CPython. We also want to detect dependencies among features automatically and compare the performance of *MoMIT* using other search algorithms, like MOEA/D-DE.

Acknowledgments

This work has been supported by the Natural Sciences and Engineering Research Council of Canada (NSERC).
Special thanks to Sami Vaarala for his valuable help with *Duktape*.

References

[1] H. Ma, L. Liu, A. Zhou, and D. Zhao, "On networking of internet of things: Explorations and challenges," *IEEE Internet of Things Journal*, vol. 3, no. 4, pp. 441–452, 2016.

[2] "Stackoverflow developer survey," https://insights.stackoverflow.com/survey/2017#top-paying-technologies, accessed: 2018-12-29.

[3] "Programming language skills (itjobswatch)," https://www.itjobswatch.co.uk/default.aspx?page=1&sortby=5&orderby=0&q=&id=900&lid=2618, accessed: 2018-12-29.

[4] "Employability tech & it skills, salary & wage analytics (worldwide)," https://gooroo.io/analytics#.XCgBxBhOmWh, accessed: 2018-12-29.

[5] M. Mossienko, "Automated cobol to java recycling," in *Software Maintenance and Reengineering, 2003. Proceedings. Seventh European Conference on*. IEEE, 2003, pp. 40–50.

[6] A. E. Hassan and R. C. Holt, "A lightweight approach for migrating web frameworks," *Information and Software Technology*, vol. 47, no. 8, pp. 521–532, 2005.

[7] T. Tonelli *et al.*, "Swing to swt and back: Patterns for api migration by wrapping," in *Software Maintenance (ICSM), 2010 IEEE International Conference on*. IEEE, 2010, pp. 1–10.

[8] M. Harman, Y. Jia, J. Krinke, W. B. Langdon, J. Petke, and Y. Zhang, "Search based software engineering for software product line engineering: a survey and directions for future work," in *Proceedings of the 18th International Software Product Line Conference-Volume 1*. ACM, 2014, pp. 5–18.

[9] N. Siegmund, S. S. Kolesnikov, C. Kästner, S. Apel, D. Batory, M. Rosenmüller, and G. Saake, "Predicting performance via automated feature-interaction detection," in *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 2012, pp. 167–177.

[10] A. S. Sayyad, J. Ingram, T. Menzies, and H. Ammar, "Scalable product line configuration: A straw to break the camel's back," in *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*. IEEE, 2013, pp. 465–474.

[11] J. Chen, V. Nair, R. Krishna, and T. Menzies, ""sampling" as a baseline optimizer for search-based software engineering," *IEEE Transactions on Software Engineering*, pp. 1–1, 2018.

[12] A. H. Ashouri, G. Palermo, and C. Silvano, "An evaluation of autotuning techniques for the compiler optimization problems," in *RES4ANT@DATE*, ser. CEUR Workshop Proceedings, vol. 1643. CEUR-WS.org, 2016, pp. 23–27.

[13] T. C. de Souza Xavier and A. F. da Silva, "Exploration of compiler optimization sequences using a hybrid approach," *Computing and Informatics*, vol. 37, no. 1, pp. 165–185, 2018. [Online]. Available: http://www.cai.sk/ojs/index.php/cai/article/view/2018_1_165

[14] D. Plotnikov, D. Melnik, M. Vardanyan, R. Buchatskiy, R. Zhuykov, and J.-H. Lee, "Automatic tuning of compiler optimizations and analysis of their impact," *Procedia Computer Science*, vol. 18, pp. 1312 – 1321, 2013, 2013 International Conference on Computational Science. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1877050913004419

[15] G. Luque and E. Alba, "Finding best compiler options for critical software using parallel algorithms," in *Intelligent Distributed Computing XII, 12th International Symposium on Intelligent Distributed Computing, IDC 2018, Bilbao, Spain, 15-17 October 2018*, 2018, pp. 71–81. [Online]. Available: https://doi.org/10.1007/978-3-319-99626-4_7

[16] K. Georgiou, C. Blackmore, S. Xavier de Souza, and K. Eder, "Less is more: Exploiting the standard compiler optimization levels for better performance and energy consumption," *CoRR*, vol. abs/1802.09845, 2018. [Online]. Available: http://arxiv.org/abs/1802.09845

[17] M. Di Penta, M. Neteler, G. Antoniol, and E. Merlo, "A language-independent software renovation framework," *Journal of Systems and Software*, vol. 77, no. 3, pp. 225–240, 2005.

[18] N. Ali, W. Wu, G. Antoniol, M. Di Penta, Y.-G. Guéhéneuc, and J. H. Hayes, "Moms: Multi-objective miniaturization of software," in *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*. IEEE, 2011, pp. 153–162.

[19] C. A. C. Coello, G. B. Lamont, and D. A. V. Veldhuizen, *Evolutionary Algorithms for Solving Multi-Objective Problems*, 2nd ed. New York: Springer, 2007.

[20] K. Deb, *Multi-objective Optimization using Evolutionary Algorithms*. Chichester: Wiley, 2001.

[21] H. Ishibuchi, N. Tsukamoto, and Y. Nojima, "Evolutionary many-objective optimization: A short review," in *2008 IEEE Congress on Evolutionary Computation (IEEE World Congress on Computational Intelligence)*, 2008, pp. 2419–2426.

[22] B. Li, J. Li, K. Tang, and X. Yao, "Many-objective evolutionary algorithms: A survey," *ACM Computing Surveys*, vol. 48, no. 1, pp. 13:1–13:35, 2015.

[23] X. Cai, Z. Yang, Z. Fan, and Q. Zhang, "Decomposition-based-sorting and angle-based-selection for evolutionary multiobjective and many-objective optimization," *IEEE Transactions on Cybernetics*, vol. 47, no. 9, pp. 2824–2837, 2017.

[24] Y. Qi, X. Ma, F. Liu, L. Jiao, J. Sun, and J. Wu, "MOEA/D with adaptive weight adjustment," *Evolutionary Computation*, vol. 22, no. 2, pp. 231–264, 2014.

[25] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," *IEEE Transactions on Evolutionary Computation*, vol. 6, no. 2, pp. 182–197, 2002.

[26] A. Zhou, B. Y. Qu, H. Li, S. Z. Zhao, P. N. Suganthan, and Q. Zhang, "Multiobjective evolutionary algorithms: A survey of the state of the art," *Swarm and Evolutionary Computation*, vol. 1, no. 1, pp. 32–49, 2011.

[27] A. Gerber. (2017) Key concepts and skills for getting started in iot. [Online]. Available: "https://developer.ibm.com/articles/iot-key-concepts-skills-get-started-iot/"

[28] ——. (2017) Choosing the best hardware for your next iot project. [Online]. Available: "https://developer.ibm.com/articles/iot-lp101-best-hardware-devices-iot-project/"

[29] V. et. al. (2019) Duktape official website. [Online]. Available: "https://www.duktape.org/"

[30] R. Morales, R. Saborido, and Y.-G. Guhneuc. (2019) MoMIT replication package. [Online]. Available: "https://moar82.github.io/momit_data/"

[31] ecma International. (2019) Sunspider javascript benchmark 1.0.2. [Online]. Available: "https://webkit.org/perf/sunspider/sunspider.html"

[32] E. Zitzler and L. Thiele, "Multiobjective evolutionary algorithms: a comparative case study and the strength pareto approach," *evolutionary computation, IEEE transactions on*, vol. 3, no. 4, pp. 257–271, 1999.

[33] C. Bonferroni, "Teoria statistica delle classi e calcolo delle probabilita," *Pubblicazioni del R Istituto Superiore di Scienze Economiche e Commericiali di Firenze*, vol. 8, pp. 3–62, 1936.

[34] A. W. Draper. (2012) Numeral.js. a javascript library for formatting and manipulating numbers. [Online]. Available: "http://numeraljs.com/"

[35] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization," *Journal of Machine Learning Research*, vol. 13, no. Feb, pp. 281–305, 2012.

[36] A. Benitez-Hidalgo, A. J. Nebro, J. Garcia-Nieto, I. Oregi, and J. Del Ser, "jmetalpy: a python framework for multi-objective optimization with metaheuristics," *arXiv preprint arXiv:1903.02915*, 2019.

[37] K.-M. Chung, W.-C. Kao, C.-L. Sun, L.-L. Wang, and C.-J. Lin, "Radius margin bounds for support vector machines with the rbf kernel," *Neural computation*, vol. 15, no. 11, pp. 2643–2681, 2003.

[38] R. K. Yin, *Case Study Research: Design and Methods - Third Edition*, 3rd ed. SAGE Publications, 2002.

[39] H. Li and Q. Zhang, "Multiobjective optimization problems with complicated pareto sets, moea/d and nsga-ii," *IEEE Transactions on Evolutionary Computation*, vol. 13, no. 2, pp. 284–302, April 2009.

Rodrigo Morales is a full-time lecturer at Concordia University in Montréal, Canada. He obtained his BS. degree in computer science in 2005 from Polytechnic of Mexico. In 2008, he obtained his MS. in computer technology from the same University, where he also worked as a Professor in the computer Science department for five years. He has also worked in the bank industry as a software developer for more than three years. He obtained his Ph.D. degree in computer engineering from Polytechnique Montréal where he earned the best thesis award of 2017. He has published in top software engineering Journals and like IEEE TSE, ESEM, and JSS and top conferences including ICSE, and SANER. He is one of the main organizers of the 1st International Workshop on Software Engineering Research & Practices for the Internet of Things (SERP4IoT), co-located with ICSE 2019, and actively participate as committee member of ICSME and ICPC conferences.

His research interests include software design quality, energy efficiency, automated-refactoring, anti-patterns, and mobile apps.

Rubén Saborido is a researcher at the Networking and Emerging Optimization group at the Department of Computer Science at University of Malaga (Spain), from 2019. He received his BS. degree in Computer Engineering and his MS. in Software Engineering and Artificial Intelligence from University of Malaga (Spain), where he worked for three years as a researcher assistant. In 2017, he received a Ph.D. in Computer Engineering from Polytechnique Montréal (Canada) and his thesis was nominated for best thesis award. In 2018 he held a postdoctoral fellowship at Concordia University (Canada), where he worked on search-based software engineering for the Internet of Things (IoT). Rubén research focuses on search-based software engineering. He is also interested in the use of metaheuristics to solve multidisciplinary real-world problems of interest for our society and computer science. He has published several papers in ISI indexed journals (such as EMSE, IEEE TSE, and Evolutionary Computation) and conference papers in IEEE ICPC, MCDM, IEEE SANER, and ACM ESEC/FSE. He has co-organized the International Conference on Multiple Criteria Decision Making, in 2013. He is on the organizing committee of the 1st International Workshop on Software Engineering Research & Practices for the Internet of Things (SERP4IoT), co-located with ICSE 2019. He is also on the application committee of the Real World Applications (RWA) track of the Genetic and Evolutionary Computation Conference (GECCO), from 2016 up today.

Yann-Gaël Guéhéneuc is full professor at the Department of Computer Science and Software Engineering of Concordia University since 2017, where he leads the Ptidej team on evaluating and enhancing the quality of the software systems, focusing on the Internet of Things and researching new theories, methods, and tools to understand, evaluate, and improve the development, release, testing, and security of such systems. Prior, he was faculty member at Polytechnique Montréal and Université de Montréal, where he started as assistant professor in 2003. In 2014, he was awarded the NSERC Research Chair Tier II on Patterns in Mixed-language Systems. In 2013-2014, he visited KAIST, Yonsei U., and Seoul National University, in Korea, as well as the National Institute of Informatics, in Japan, during his sabbatical year. In 2010, he became IEEE Senior Member. In 2009, he obtained the NSERC Research Chair Tier II on Software Patterns and Patterns of Software. In 2003, he received a Ph.D. in Software Engineering from University of Nantes, France, under Professor Pierre Cointe's supervision. His Ph.D. thesis was funded by Object Technology International, Inc. (now IBM Ottawa Labs.), where he worked in 1999 and 2000. In 1998, he graduated as engineer from école des Mines of Nantes. His research interests are program understanding and program quality, in particular through the use and the identification of recurring patterns. He was the first to use explanation-based constraint programming in the context of software engineering to identify occurrences of patterns. He is interested also in empirical software engineering; he uses eye-trackers to understand and to develop theories about program comprehension. He has published papers in international conferences and journals, including IEEE TSE, Springer EMSE, ACM/IEEE ICSE, IEEE ICSME, and IEEE SANER. He was the program co-chair and general chair of several events, including IEEE SANER'15, APSEC'14, and IEEE ICSM'13.