2018 Third ACM/IEEE Symposium on Edge Computing

EveryLite: A Lightweight Scripting Language for Micro Tasks in IoT Systems

Zhenying Li*†, Xiaohui Peng*, Lu Chao*†, Zhiwei Xu*
*Institute of Computing Technology, Chinese Academy of Sciences
†University of Chinese Academy of Sciences
Beijing, China
{lizhenying18s, pengxiaohui, chaolu, zxu}@ict.ac.cn

***Abstract*—Processing the computational tasks on the devices at the edge can significantly reduce computing load, network transmission load, and response latency. However, programming on these devices is difficult due to the resource-constrained and diversity features. This paper presents a lightweight scripting language, called EveryLite, to address this issue. EveryLite features a new @-expression to access the resources on connected devices via the REST Web interfaces and focuses on the micro tasks with limited complexity in Internet of Things (IoT) systems. We design an elastic runtime environment with a core of 37 KB and some extending modules to address the IoT devices' diversity problem. Experimental results show that the applications developed by EveryLite can be run on heterogeneous devices without modification and consume less memory than those developed by other scripting languages such as Lua and Python.**

***Index Terms*—Scripting language, Web of Things, resource-constrained, computation offloading, micro tasks.**

I. Introduction

The proliferation of IoT [1] creates massive data on devices and network edges. New computing paradigms such as edge computing [2] are proposed with the vision of offloading computing tasks from the cloud to the edge and performing computing near the data source. They significantly reduce the network transmission, response latency, and energy consumption. Abundant applications are needed for moving tasks from cloud to IoT devices at the edge. However, programming at the edge is very difficult due to the resource-constrained and diversity features of the IoT devices. These devices form a heterogeneous distributed platform at the network edge in which the nodes are different from each other, making it necessary for developers to have a detailed understanding of the devices' system architectures. This brings great challenges for programming on the IoT devices and increases the difficulty of implementing computation offloading in which the applications should be freely offloaded among all the devices. Meanwhile, most of the IoT devices are resource-constrained, so the applications should be carefully designed with the concerns of low latency, low power consumption, small footprints, etc. Programmers should be fully experienced and understand the know-how of developing the system for IoT devices.

We observed that a large number of computation offloading tasks on resource-constrained devices are lightweight such as collecting the data from the sensors, doing the Extraction-Transformation-Loading (ETL) operations on the data, and neural network inferring. Specifically, we call the above lightweight tasks as *micro tasks* in IoT ecosystems, if the task is executable in a device and satisfies the device's resource constraints and user experience significantly. The potential offloading targets of micro tasks are resource-constrained devices such as sensor devices and wearable devices. Most resource-constrained IoT devices' ROM and RAM sizes are usually several hundred KB [3]. The IoT devices are also diverse in the hardware architectures and the operating systems, for example, there are dozens of operating systems such as Contiki [4], Micro T-kernel [5], TinyOS [6], and EdgeOS_{H} [7].

The diversity of IoT devices makes the applications that developed with static programming languages such as C and C++ and compiled for a single target difficult to be offloaded freely. The program that implements computation offloading should be independent of the underlying system API, and the runtime environment needs to be small. Scripting languages are usually interpreted at runtime, which is better for offloading applications to another device. However, most of them are designed for the platforms with rich computing resources and the unified architecture such as mobiles and PCs. They usually have virtual machines or complex engines that hide the system details and cannot be run on resource-constrained devices due to the large footprints.

This paper presents EveryLite, a lightweight scripting language, which focuses on composing micro tasks on IoT devices. The EveryLite language and its runtime environment can minimize the underlying dependencies on the heterogeneous systems by using the @-expression for accessing the Web resources. We also implement an extensible and portable runtime environment by using modular and layered design principles to address the diversity problem.

The contributions of this paper are listed as follows: 1) A lightweight scripting language which supports RESTful resource manipulation methods is proposed to hide the access details of system-level resources to ease the programming procedure for micro tasks. 2) A small footprint runtime environment core with 37 KB is implemented which indicates that it provides the runtime support for extremely resource-

978-1-5386-9445-9/18/$31.00 ©2018 IEEE
DOI 10.1109/SEC.2018.00050

constrained devices. 3) Modular and layered design principles are applied to add elasticity to the runtime environment and to decouple the system dependencies from the application logic.

II. Design and Implement

A. Design Overview

To address the programming challenges on resource-constrained diverse IoT devices and compose micro tasks for edge systems, we adopt the following principles to design the EveryLite language and its runtime environment.

The EveryLite language is designed with the following concerns. 1) Execute application codes on heterogeneous devices with the least modifications. When accessing out-of-scope data or variables, such as reading historical data records from databases, resource variables are recommended to fully isolate the diversity of interfaces instead of calling the low-level APIs of the operating system. However, invoking the out-of-scope APIs is still permitted in the worst cases while sacrificing the program portability. 2) Support micro tasks. EveryLite does not pay attention to complex tasks such as multithreading, heavy invoking stacks or daemon process which may excess the computing capacities and energy constraints of devices.

The runtime environment is designed with the following two basic principles. 1) Small footprint. The EveryLite runtime environment should have a small footprint by applying a lightweight garbage collection strategy, minimizing built-in functions and using extensible modules which can be configured according to the computing resources on devices. 2) High portability. We propose a layered design to decouple the operating system kernel from the application business logic by introducing a system abstraction layer and limiting the porting complexity to a few POSIX interfaces only.

Figure 1 shows the architecture of the EveryLite runtime environment. From the vertical view, we limited the interfaces between the EveryLite runtime and various underlying systems by implementing a system abstraction layer, which tries to address the portability problem among heterogeneous devices. From the horizontal view, we allow switching off unnecessary modules to make the trade-off between the memory footprint and functionality. In the worst case, the runtime environment can be reduced to the core module only.

B. Micro Task

The micro task is the executable task that satisfies the resource constraints and the user experience significantly for a specific device. To satisfy the device's resource constraints, the micro task may have to consume little energy and have a small footprint in terms of the battery and the memory limitations of the device. The micro task may also have a low response latency compared with the user's expectation to satisfy the user experience. A micro task needs to satisfy the constraints of the response latency, the memory and disk space, and the energy consumption, so we can use the limitation tuple <time, memory, energy> to judge if a task is a micro task.

The micro task is a relative concept that varies on the specific target executing devices. For example, the inference of a ten-layer convolutional neural network is a micro task in a PC because it can be finished usually under 1 ms and has a small footprint compared with the PC's GB-level memory, but it is not a micro task in the sensors with KB-level memory that is not enough to execute the application.

The micro task also depends on the user experience. For example, a task with 1 second execution time but expected to be responded in 10 ms is not a micro task. However, if the user expectation on time can be loosed to 1 second or higher, the task will be a micro task.

C. The Syntax of EveryLite

The syntax of EveryLite is similar to Lua [8] which is a lightweight scripting language with a high execution efficiency. However, the footprint of the Lua interpreter exceeds 100 KB which indicates that it cannot be executed on many extremely resource-constrained IoT devices.

1) Basic Syntax: The basic syntax of EveryLite inherits from Lua, providing the fundamental functions such as arithmetic and logic operations, and function calls. The developer can also write EveryLite functions which can be used as the mobile code in the computation offloading.

2) @-expression: To reduce the dependency on the underlying systems, EveryLite provides a unified method of accessing the device resources by using resource variables and the @-expression. The resource variable is the representation of the RESTful URI, for example, the variable $temperature$ may stand http://10.0.0.3/temperature?t=[x]&p=bedroom. The developers use the @-expression, which contains the resource variable name, the symbol @, and a set of parameters, to read or update the resource. The operations of resource variables are listed as follows:

- v = resource_id@$(p_1, ..., p_n)$. It reads the resource named $resource\_id$ under the condition that consists of parameters $p_1, ..., p_n$.
- resource_id@$(p_1, ..., p_n)$ = v. It updates the $resource\_id$ with the value v under the parameters $p_1, ..., p_n$.

In the example of the $temperature$, the value of the parameter t in the URI is flexible and is usually determined in the runtime. While the transfer protocol, the host address, the path and query string are relatively stable, they will only be modified when the codes are moving among the devices. The accessing logic in a specific application is stable, and it can be executed on various devices without modifying the code. EveryLite decouples the flexible part as the parameters in the @-expression and the relatively stable parts in the background configuration with the resource variables. With the @-expression, we can access the $temperature$ with the code *t = temperature@(x)*. As shown in figure 2, the accessing URIs for the same type of functions are slightly different in their protocols and addresses brought by multiple vendors and device locations. Traditionally, the source code of the application needs to be modified when the device's host address changes. The EveryLite code only contains the data access logic and hides the accessing methods with the @-expression, and the

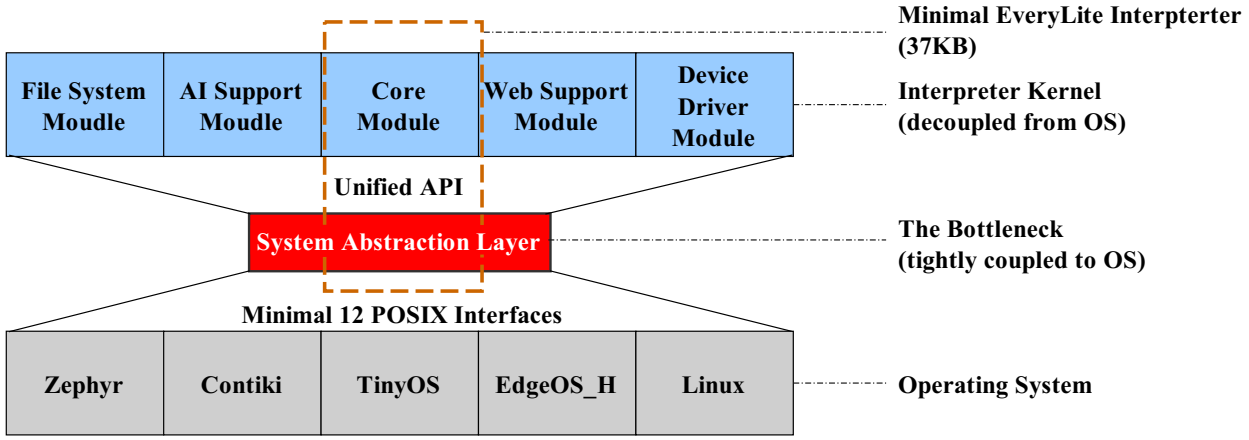

Fig. 1. Architecture overview.

developer could possibly write one scripting application to support multiple similar scenarios.

D. The Interpreter

The core of the runtime environment is the scripting interpreter. This section presents the overview of the EveryLite interpreter's design.

1) System Abstraction Layer: To improve the interpreter's portability, we implement a system abstraction layer to address the diversity issue of IoT devices. System abstraction layer encapsulates the device-related APIs and provides the unified interfaces for the interpreter kernel. It is built on a minimum of 12 POSIX interfaces which make EveryLite's interpreter portable.

2) Modular Design: Due to the resource-constrained and diversity features of the devices, the EveryLite interpreter adopts a modular design style. The complete interpreter consists of five modules shown in figure 1. The core module provides the fundamental functions for executing the EveryLite code. The file system module encapsulates the file system APIs, and the interpreter can be run on the devices without file systems by closing its file system module. The Web support module encapsulates four functions which perform the 4 primitive operations: PUT, POST, GET and DELETE in HTTP [9]. The AI support module provides APIs for handling the neural network tasks. The driver module encapsulates the driving functions such as reading temperature sensor API on the smart thermometer.

When deploying the interpreter on the IoT devices, the developer can choose appropriate modules according to the device specification and the computation task. The following macros are used to include or exclude each module in the compiling stage.

```
#define FS_MODULE       1
#define AI_MODULE       0
#define WEB_MODULE      0
#define DR_MODULE       0
```

Except for core module, the APIs in the other four modules can be configured by the developers according to the device's ability. In EveryLite, the developers need to add the C function pointers and names to the list of library functions in the header files for each module.

3) Resource Handler: As aforementioned, the resource variables and @-expression are designed to simplify and unify the Web resource manipulations in EveryLite. EveryLite provides the developers with a configurable resource handler to support the resource variables and @-expression which hide the resource accessing details. The resource handler needs to know how to obtain the URI of the remote resources. The format of the resource handler is shown as follows:

```
resource_handler(res_name, v, p, method)
```

The resource handler requires four parameters. The res_name is the name of the resource variable, and v is the value of the resource. The p composes of the parameters used for reading and updating the resource. The $method$ indicates one of the 4 HTTP methods, and we defined them as RES_PUT, RES_GET, RES_POST, and RES_DEL in EveryLite. The resource handler returns zero indicating success and non-zero indicating an exception.

4) Lightweight Garbage Collection: To reduce the size of the runtime environment, EveryLite uses a lightweight garbage collection strategy. The garbage collection in EveryLite interpreter will not be performed until the program finishes running and then all objects using dynamic memory space will be released. EveryLite only focuses on micro tasks, so even if garbage collection is not performed during the programs running, the interpreter can still perform the tasks normally in most cases. Meanwhile, the interpreter records the dynamic memory size used by the program, and once it exceeds the preset threshold, the interpreter terminates the program and returns an error code indicating that the memory space used is too large. Because of the micro tasks and the memory space check, the EveryLite interpreter uses a lightweight garbage collection instead of a complex garbage collection strategy

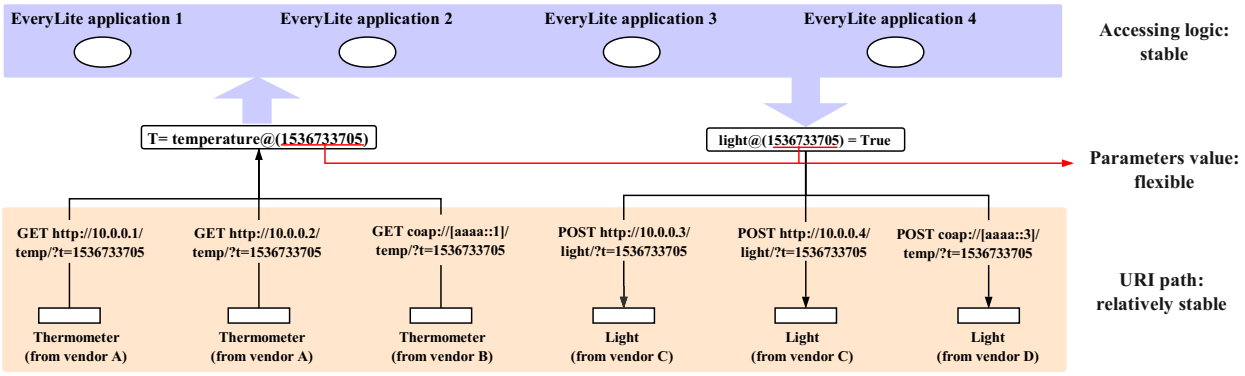

Fig. 2. Access the devices with different URIs using the same EveryLite code.

which not only produces a larger footprint but also reduces the efficiency.

III. Evaluation

In this section, we introduce 3 experiments to show the merits of EveryLite and its runtime environment for micro tasks on resource-constrained IoT devices. Experiment III-A and III-B evaluate the performance and footprint of the EveryLite interpreter respectively and III-C examines its support for computation offloading.

A. *Efficiency*

In this experiment, we compare EveryLite with C, Lua, Python, mRuby [10], and JerryScript [11]. EveryLite inherits the basic design from Lua. Python is a powerful high-level programming language that is widely used. JerryScript implements a lightweight execution engine of JavaScript and mRuby is a lightweight version of Ruby. C programming language is one of the most efficient one in the high-level languages for IoT devices and thus is selected as a baseline in this experiment. The C programs in the experiment are compiled with GCC 4.9.2, and the versions of Lua, Python, mRuby, and JerryScript are 5.3.1, 2.7.1, 1.4.1, and 1.0.0 respectively.

The experiment is conducted on a Raspberry Pi 2, and we developed four programs: *Print*, *Branch*, *Loop*, and *Matrix multiplication* in the six languages respectively. In *Print* test, programs output a string to the screen by doing driving function calls and I/O operations. *Branch* test is to select the state of the air conditioner according to the temperature. *Loop* calculates the value of the harmonic series from 1 to 100 million to test the efficiency of doing simple computation on large amounts of data. *Matrix multiplication* is developed to examine the performance of programs when doing matrix multiplication.

We run this four sets of programs 100 times and record the execution time respectively. Figure 3 shows the experimental results. In *Print*, *Branch*, and *Loop*, EveryLite receives the best performance among the scripting interpreters. Especially in the *Print*, the execution time of the EveryLite program is only 20% higher than the C program. On average, the

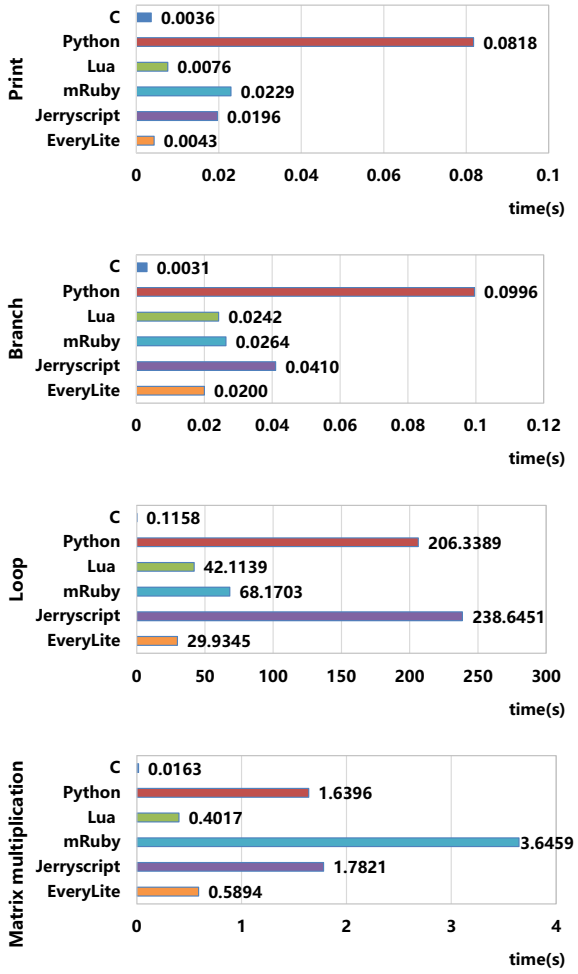

Fig. 3. The execution time of the programs in different languages.

execution time of the programs written in EveryLite is 77%, 74%, 21% and 88% lower than that of JerryScript, mRuby, Lua, and Python respectively. Result shows that EveryLite language and its interpreter can handle the micro task more efficiently.

B. Footprint

Footprint is one of the key concerns for resource-constrained devices. In this section, we examine the footprints of the five interpreters and the C program when performing the $Print$ test in III-A on a desktop computer and analysis the breakdown memory occupations of them. We evaluate their footprints by the size of used dynamic memory and the compiled binary files.

Table I presents the size of the dynamic memory space and the compiled binary files of these scripting interpreters and the C program. Figure 4 shows the total memory of the different implementations when performing $Print$ test. Among the scripting interpreters, the memory used by the EveryLite is 21.9%, 1.5%, 20.3%, 1.2% of that used by JerryScript, mRuby, Lua and Python respectively. The footprint experiment shows that the EveryLite interpreter has the smallest footprint among these common scripting languages, therefore can be run on more IoT devices.

C. Computation Offloading

In this section, we set up a scenario for evaluating the computing offloading using EveryLite. The experiment environment consists of a desktop computer and an IoT device equipped with a ΦPU [12] and a camera. They are connected through the network. The task is capturing images using the camera and identifying the digits in the images. In the traditional processing mode, PC sends the request to the camera to obtain the image and runs the recognition program to recognize the digit. In our computation offloading mode, PC sends the EveryLite code, which uses the $eval\_nn$ function and the AI support module to compute the neural network load, to the IoT device. Finally, the recognition is done on the IoT device and the device sends the result back to the PC.

In this experiment, the following EveryLite function $digit$ is used to identify the digits from captured images. The $digit$ function uses the @-expression to require resources and uses the $load\_nn$ and $eval\_nn$ functions, which have been decoupled from the devices due to the system abstraction layer, to perform the digit recognition task so that the code can be migrated from PC to the camera without modification.

TABLE I
THE MEMORY USAGES OF DIFFERENT IMPLEMENTATIONS

Implementation	Heap(KB)	Stack(KB)	Dynamic Memory(KB)	Binary File(KB)
EveryLite	3.45	3.02	6.06	37.30
JerryScript	0.00	1.27	1.27	196.50
mRuby	177.13	3.18	180.30	2682.97
Lua	16.27	11.35	27.62	185.77
Python	512.80	19.73	523.80	3082.47
C	0.00	2.25	2.25	7.17

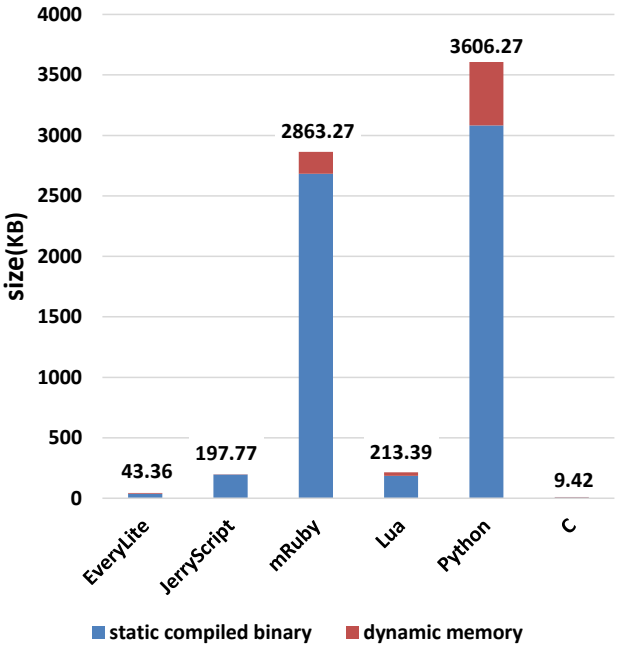

Fig. 4. The memory usages of different implementations.

```
function digit (t)
    image = CAM_IMAGE @(t)
    nn = load_nn(NN@("ann"))
    DIGIT@(t) = eval_nn(image, nn)
end
```

Listing 1. The function of identifying digits.

Result shows that in the traditional process mode, more than 95% of the time was spent on the network transmission. On the contrary, the communication throughput in the computation offloading mode decreased from 150 KB to 78 B, resulting in a 97.6% reduction in the response time which may improve the user experience significantly for time-sensitive tasks.

IV. Related-work

As high-level dynamic programming languages, scripting languages tend to be more portable and flexible. Combining with resource-constrained devices, researchers have explored and reconstructed new designs over existed scripting languages such as JavaScript, Lua, and Ruby, to break through the obstacles brought by emerging runtime scenarios.

JerryScript [11] implements a lightweight execution engine of JavaScript. It is launched by Samsung and written in C language with the C99 standard to improve portability. JerryScript adopts the compressed pointers that use 16-bit values to represent 8 byte aligned addresses, resulting in saving 50% of memory on 32-bit systems, uses a more aggressive garbage collection strategy and removes the abstract tree to reduce the footprints.

eLua [13] is an implementation of Lua for embedded devices. It defines the concept of $platform\ interface$ which

TABLE II
FEATURES COMPARISON OF RELATED PROGRAMMING LANGUAGES

	Footprint	Portability	Performance	OS Friendliness	Functionality
mRuby	Large (>512KB)	Medium	Low	High	High
JerryScript	Medium (<256KB)	Medium	Medium	High	High
eLua	Small (<64KB)	Low	High	Low	Medium
EveryLite	Small (<64KB)	High	High	High	Low

tries to group the common attributes of different platforms to improve the program's portability. To reduce the footprints, eLua uses $lightfunctions$, which can't have upvalues or environments, and $rotables$, the read-only tables that can be read directly from ROM and needn't use the RAM space, to reduce the footprints.

mRuby [10] is a lightweight Ruby for embedded software and resource-constrained devices. It also uses the C99 standard and removes the file input and output section to run on the platforms without the file system. mRuby prunes the classes that are not providing the core functions from libraries and removes the parser section to get a smaller footprint.

These implementations have advanced in reducing footprints and increasing portability, but they still have problems. As shown in table II, the footprint of mRuby even exceeds 512 KB with the result that it cannot be run on many resource-constrained IoT devices. JerryScript needs to be developed further in portability and performance, and eLua can only run on the bare hardware, so developers need to have a detailed understanding of the hardware layer of the device. Focusing on micro tasks only, EveryLite compromises on the functionality but raises an extremely low footprint, high portability and performance instead.

V. CONCLUSION

This paper presents a lightweight scripting language EveryLite to address the programmability on the resource-constrained IoT devices which are diverse in architectures and systems with computation offloading scenarios. EveryLite focuses on composing the micro tasks and provides the @-expression to access the Web resources, which is also designed to be lightweight, extensible and portable by using the modular and layered design principles. We further evaluate the performance and the footprint of the EveryLite runtime environment and validate the mobility of codes in an image recognition scenario with computation offloading.

EveryLite outperforms the scripting languages tested in the experiment. The execution time of the EveryLite application is 77% and 74% lower than JerryScript and mRuby on average, respectively. Even compared with optimal C language, EveryLite brings only 20% running-time overhead in the $Print$ test.

EveryLite is also the most lightweight implementation among these scripting language runtime environments. The compilation size of the EveryLite interpreter is 37 KB in the minimum case, which is 18.9% and 1.4% of JerryScript and mRuby. The EveryLite interpreter uses the least size of the memory among the different scripting implementations compared in this paper, which shows that it is more suitable for resource-constrained devices.

EveryLite code can be offloaded with almost no modification. In the experiment, we offload the computation from a server to a camera to perform AI inference task using the EveryLite code, and the application can be run on both the server and the camera without modifying any code. It reduces 97% whole execution time on the camera than on the server.

ACKNOWLEDGMENT

This work is supported by the Key Program of the National Natural Science Foundation of China under Grant No.61532016 and the CAS Pioneer Hundred Talents Program under Grant No.Y704061000.

REFERENCES

[1] K. Ashton, "That 'internet of things' thing," *RFID Journal*, vol. 22, no. 7, pp. 97–114, 2009.
[2] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE Internet of Things Journal*, vol. 3, no. 5, pp. 637–646, 2016.
[3] X. Peng, X. Zhang, Y. Wang, and L. Chao, "Web enabled things computing system," *Journal of Computer Research and Development*, vol. 55, no. 3, pp. 572–584, 2018.
[4] A. Dunkels, B. Gronvall, and T. Voigt, "Contiki-a lightweight and flexible operating system for tiny networked sensors," in *Proceedings of International Conference on Local Computer Networks*. Tampa, FL, USA: IEEE, 2004, pp. 455–462.
[5] M. Kamio, K. Nakamura, S. Kobayashi, N. Koshizuka, and K. Sakamura, "Micro t-kernel: A low power and small footprint rtos for networked tiny devices," in *Proceedings of International Conference on Information Technology: New Generations*. Las Vegas, NV, USA: IEEE, 2009, pp. 587–594.
[6] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo *et al.*, *TinyOS: An operating system for sensor networks*. Springer, 2005.
[7] J. Cao, L. Xu, R. Abdallah, and W. Shi, "Edgeos_h: A home operating system for internet of everything," in *Proceedings of the International Conference on Distributed Computing Systems*. Atlanta, GA, USA: IEEE, 2017, pp. 1756–1764.
[8] R. Ierusalimschy, L. H. De Figueiredo, and W. C. Filho, "Lua-an extensible extension language," *Software: Practice and Experience*, vol. 26, no. 6, pp. 635–652, 1996.
[9] R. T. Fielding and R. N. Taylor, "Principled design of the modern web architecture," *ACM Transactions on Internet Technology*, vol. 2, no. 2, pp. 115–150, 2002.
[10] A. D. Nagumanthri and K. Tanaka, "Internet of things with mruby," in *Proceedings of the International Conference on Information Technology*. Noida, India: IEEE, 2016, pp. 142–147.
[11] E. Gavrin, S.-J. Lee, R. Ayrapetyan, and A. Shitov, "Ultra lightweight javascript engine for internet of things," in *Proceedings of the SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity*. Pittsburgh, PA, USA: ACM, 2015, pp. 19–20.
[12] Z. Xu, X. Peng, L. Zhang, D. Li, and N. Sun, "The ϕ-stack for smart web of things," in *Proceedings of the Workshop on Smart Internet of Things*. San Jose, California, USA: ACM, 2017, pp. 1–6.
[13] eLua project. [Online]. Available: http://www.eluaproject.net/