Performance and Resource Analysis on the JavaScript Runtime for IoT Devices

Dongig Sin and Dongkun Shin(✉)

Department of Electrical and Computer Engineering, Sungkyunkwan University, Suwon, Korea
{dongig,dongkun}@skku.edu

Abstract. The light-weight JavaScript frameworks such as IoT.js, DukServer, and Smart.js provide the asynchronous event-driven JavaScript runtime for low-end IoT device. These frameworks are designed for memory-constrained systems such as IoT devices. To evaluate the performance of these frameworks, existing JavaScript benchmarks are not suitable considering that the use cases of IoT device are mainly to execute a simple task generating sensor and network I/O requests. In this paper, we propose several IoT workloads to evaluate the performance and memory overhead of IoT systems, and evaluate several light-weight JavaScript frameworks. In addition, we evaluated the effectiveness of multi-core system for JavaScript framework.

Keywords: JavaScript engine · Internet of Things · Low-memory · Server-side JavaScript framework · Iot platform

1 Introduction

The JavaScript (JS) programming language is widely used for web programming as well as general purpose computing. JavaScript has the advantages of easy programing and high portability due to its interpreter-based execution model. Recently, various event-driven JavaScript frameworks have been introduced such as Node.js, which is based on Google V8 JavaScript engine and the libuv event I/O library [1]. To distinguish from the JS engine of web browser, these event-driven JS frameworks are called server-side JS framework since they can be used for implementing server computers.

The event-driven JavaScript environment is useful for implementing IoT systems which should handle many sensor and network I/O events. However, Node.js is designed for server computer system, and thus it is not optimized for resource usage. Therefore, Node.js is not suitable for IoT devices which have low-end processors and limited memory space in order to reduce product cost and power consumption. Accordingly, various light-weight JavaScript engines requiring small footprint and runtime memory have been proposed, such Duktape [2], JerryScript [3], and V7 [4].

In this paper, we propose several IoT workloads and evaluate the performance and resource usage of various event-driven JavaScript runtimes.

DOI: 10.1007/978-3-319-42085-1_50

2 Related Work

Recently, there have been several efforts trying to analyze performance of the server-side JavaScript framework for server computer system. Ogasawara [5] conducted context analysis of server-side JavaScript applications on the Node.js, and found that little time is spent on dynamically compiled code. Zhu et al. [6] present an analysis of the microarchitectural bottlenecks of scripting-language-based server-side event-driven applications. In addition, much work has been done to analyze and to improve the performance of JavaScript engine used in web browsers [7–10].

The prior works did not focus on the server-side JavaScript framework for memory-constrained devices and the lightweight JavaScript engine. Our work studies the lightweight server-side JavaScript framework and handles the workloads for the IoT environment, not for the web browser.

3 Light-Weight JavaScript Engine

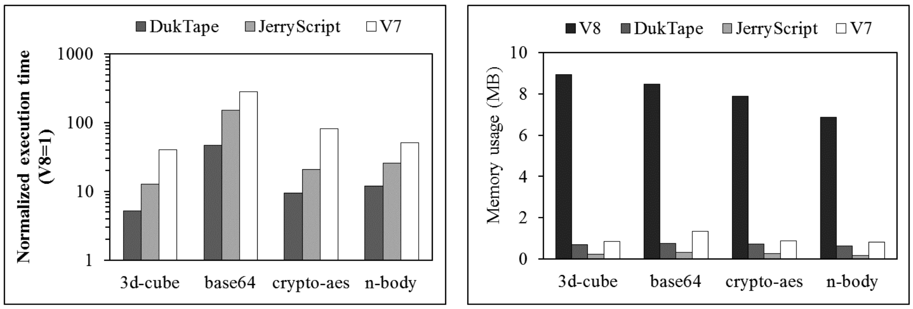
The JavaScript engines for web browsers generally use the just-in-time (JIT) compiler technique for high performance. For instance, V8 engine [11] monitors the execution frequencies of JavaScript functions compiled by the base compiler at runtime. If a function is frequently executed, it is compiled by the optimizing compiler, called Crankshaft, through the techniques of hidden class and inline caching. However, these optimization techniques are not suitable for low-end IoT devices since they require a large amount of memory.

Therefore, several light-weight JavaScript engines are introduced, which support only the minimal functions of JavaScript language following ECMA standard. Considering memory-hungry IoT devices, the size of heap memory should be limited and aggressive garbage collection is necessary. As shown in Table 1, the binary file sizes of light-weight JavaScript engines are up to 125 times less than that of V8 engine.

Table 1. Binary file size of JavaScript engine

	V8	Duktape	JerryScript	V7
Size (KB)	21504	192	172	1228

Figure 1 shows the performances and memory consumptions of several JavaScript runtimes while executing SunSpider benchmark. V8 provides up to 300 times better performance compared with the light-weight JS engines. However, the memory consumption of V8 is significantly higher than those of light-weight JS engines. This is because V8 allocates a large memory pool for high performance. On the contrary, the performances of light-weight JS engines are significantly low since they do not adopt the JIT optimizing compiler. However, they require only several mega-bytes of run-time memory since the heap memory is limited and the aggressive garbage collection reclaims unused object memory within a short time interval. The memory optimization techniques further degrade the performance.



(a) Execution time **(b) Memory usage**

Fig. 1. Execution time and memory usage of SunSpider benchmark

However, the SunSpider benchmark is designed to measure the performance of JS engine for high-end devices, and thus it includes computing and memory-intensive applications such as 3D cube rotation or encoding/decoding [12]. These applications are not suitable for measuring the performance of IoT device which handles only sensor data and network I/O requests. Consequently, in order to evaluate the performance and memory overhead of light-weight JS engine, we need IoT-specific workloads.

Based on JS engines, several event-driven JS frameworks are announced. Node.js is a platform built on V8 JS engine for easily building network applications. Node.js uses an event-driven, non-blocking I/O model that is useful for data-intensive applications. It is possible to extend the functionality of Node.js via package module called NPM. Many open source modules are available for NPM.

Although Node.js is versatile and it is based on high-performance V8 JS engine, it is inadequate for resource-constraint IoT devices. In this paper, we focus on light-weight JS frameworks for IoT devices. IoT.js has a similar architecture with Node.js [13]. However, it uses a light-weight JS engine called JerryScript. IoT.js also supports the package module. However, only a small number of packages are available currently. DukServer uses the Duktape JS engine. It supports C socket-based communication, with which DukServer can communicate other native functions. Therefore, the external native functions should be integrated with DukServer at build time. Currently, only the HTTP-server application is included at DukServer. Smart.js, relying on V7 engine, supports the binding to the network and hardware native API. In addition, the device firmware can be called from JS applications for bare metal execution.

4 Experiments

4.1 Experiment Environments

The HTTP servers are implemented with Node.js, IoT.js, DukServer and Smart.js. The hardware is ODROID-U3 which has 1.7 GHz Quad CPU and 2048 MB memory. We evaluate the performance and memory overhead of JS framework while running several IoT workloads. Four IoT service workloads are used as shown in Table 2. The workloads are designed considering the common scenarios of IoT systems.

Table 2. Workload specification

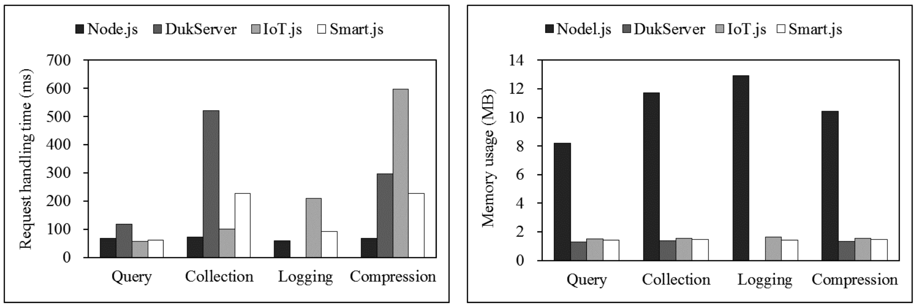
Name	Business logic	Feature
Query	Send a specific sensor data	Usual case
Collection	Transmit collected sensor data after sorting	Data-intensive
Compression	Transmit collected sensor data after compressing	Computing-intensive
Logging	Save sensor data to file	I/O-intensive

4.2 Performance and Memory Usage of IoT Workload

For performance comparison, the request handling times of the HTTP server implemented by JS framework are measured while client-side applications sends several requests to the server. In every workload, the first request handling time of Node.js is long compared with the following request handling times since the JS operations are not optimized at the first execution.

4.2.1 Query Workload

In IoT system, the most general scenario is to transfer specific sensor data requested by client. The Query workload reads sensor data and transfers them to client. Since the workload has no computing-intensive job, Node.js cannot benefit from the optimizing compiler of V8 JS engine as shown in Fig. 2(a). It shows rather performance degradation due to additional code execution for optimization. DukServer also shows longer request handling times than other frameworks. While V7 and JerryScript engines use a memory pool to assign a memory space for object, Duktape allocates the memory on-demand. Therefore, there is a memory allocation overhead whenever an object is created, even though the overall memory usage is lower than other schemes.



(a) Request handling time of IoT Workloads **(b) Memory usage of IoT workloads**

Fig. 2. Performance and memory usage of IoT workload

4.2.2 Collection Workload

If an IoT device collects data from multiple sensors and processes them, a large size of memory space is required for the collected sensor data. The Collection workload collects, sorts, and transfers two hundreds of sensor values. For the sorting operation, many memory pages should be allocated. The performance of Collection workload is greatly affected by memory management technique of JS framework. Duktape has a high memory allocation overhead since it does not maintain memory pool, as shown in Fig. 2(a). JS framework needs a memory reclamation technique. There are two kinds of techniques used by current JS engines. The reference counting technique maintains the reference count of each object, and de-allocates the memory space of unused objects if the reference counts of them are zero. The second technique is garbage collection (GC) which is triggered when there is memory pressure. Once the GC is triggered, the mark-and-sweep operation is performed, which marks only referenced objects and then frees unmarked objects. Duktape uses both of the techniques, and saves the frequently accessed objects at hash table in order to reduce the search cost of garbage collection. On the other hand, V7 uses only the garbage collection, and manages the objects in a linear list, which causes a high search cost of garbage collection. Therefore, Smart.js shows a long request handling time due to the garbage collection cost.

4.2.3 Logging Workload

The IoT device can store the collected sensor data at storage device via file systems. The Logging workload stores two hundreds of sensor data at file system. Figure 2(a) shows that the request handling time of IoT.js is longer than other frameworks. The native file I/O operations are called by the JS applications via a native API binding technique. Since the native API binding techniques provided by each framework is different, the file I/O operations show different performances. The write API provided by IoT.js allocates a buffer object in order to transform the target data to a common type. Therefore, IoT.js shows poor performance in the file I/O intensive workload due to the overhead of buffer allocation and data transformation. In this workload, DukServer is excluded since it does not support the file system module.

4.2.4 Compression Workload

The sensor data collected by IoT device can be compressed for fast network transfer. The Compression workload transfers a hundred of sensor values after compressing them with the LZW algorithm. The light-weight JS frameworks show poor performances for the computing-intensive workload compared with Node.js.

Although the light-weight JS frameworks show worse performances than Node.js in most of the IoT workloads, the performance gap is less than several hundreds of milliseconds. However, the memory consumptions of light-weight JS frameworks are significantly lower than that of Node.js as shown in Fig. 2(b). Therefore, these frameworks are suitable for low-end IoT devices.

4.3 Performance on Multi-core Architecture

Recent embedded systems adopt multi-core processors for high performance. However, the single thread-based JavaScript framework cannot benefit from multi-core systems [6]. Moreover, multiple processor cores will cause high power consumption. Since the power consumption is also important metric for IoT devices, the power efficiencies of JS frameworks are observed.

For experiments, ODROID-XU3 is used, which has ARM Cortex A15 2.0 GHz Quad CPU and A7 1.5 GHz Quad CPU, called big.LITTLE architecture. The high-performance big cores support the out-of-order execution and use a high CPU clock. The low-performance LITTLE cores perform the in-order execution with a low CPU clock. We measure the request handling time while running the Compression and Logging workloads as shown in Fig. 3.

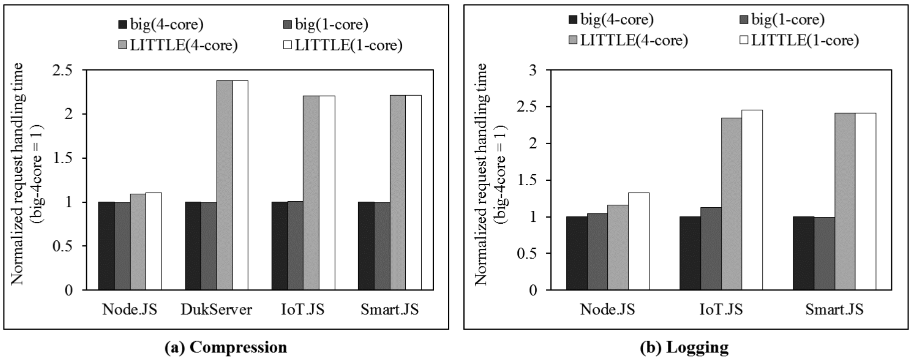


Fig. 3. Performance on multi-core system

For the Compression workload, the request handling times of light-weight JS frameworks are significantly reduced on the high-performance cores. However, there are no performance changes by the number of enabled cores since the JS frameworks use the single-thread execution model. The performance improvement by high-performance cores is insignificant at Node.js due to its optimizing compiler.

For the I/O-intensive Logging workload, the multi-core architecture improves the performance at Node.js and IoT.js. These frameworks support the asynchronous write API which uses I/O thread pool to prevent the main-thread to be blocked during the I/O operation [14]. Therefore, multiple cores can execute the main thread and I/O threads simultaneously. The multi-core processor improves the I/O performance of Node.js and IoT.js up to 12 % compared with the single-core system. On the contrary, Smart.js handles the I/O operations in the single main thread, and thus its performance is not improved by multi-core processor.

Although the multi-core processor can improve the performance of JS frameworks which use separated I/O threads, the performance improvement is not significant. Moreover, the multi-core processor can waste power without any performance improvement on computing-intensive workloads. Therefore, the single-core system with high performance will be more efficient than the multi-core system for IoT systems considering both performance and power.

5 Conclusion

This paper proposed several representative IoT workloads to evaluate the performance and memory overhead of IoT devices. We analyzed the features of different light-weight JavaScript frameworks with the IoT workloads. In addition, we evaluated the effect of multi-core system for JavaScript framework. Our analysis results will be useful for selecting or designing light-weight JavaScript framework and hardware systems for IoT applications.

Acknowledgements. This work was supported by the Center for Integrated Smart Sensors funded by the Ministry of Science, ICT & Future Planning as Global Frontier Project. (CISS-2011-0031863).

References

1. Node.js. https://nodejs.org
2. Duktape. http://www.duktape.org
3. JerryScript. https://samsung.github.io/jerryscript
4. V7. https://www.cesanta.com/developer/v7
5. Ogasawara, T.: Workload characterization of server-side javascript. In: Proceedings of IEEE International Symposium on Workload Characterization (IISWC) 2014, pp. 13–21. IEEE (2014)
6. Zhu, Y., Richins, D., Halpern, M., Reddi, V.J.: Microarchitectural implications of event-driven server-side web applications. In: Proceedings of the 48th International Symposium on Microarchitecture, pp. 762–774. ACM (2015)
7. Chadha, G., Mahlke, S., Narayanasamy, S.: Efetch: optimizing instruction fetch for event-driven web applications. In: Proceedings of the 23rd International Conference on Parallel Architectures and Compilation, pp. 75–86. ACM (2014)
8. Zhu, Y., Reddi, V.J.: WebCore: architectural support for mobile web browsing. In: Proceedings of International Symposium on Computer Architecture, p. 552 (2014)
9. Anderson, O., Fortuna, E., Ceze, L., Eggers, S.: Checked load: architectural support for JavaScript type-checking on mobile processors. In: Proceedings of 2011 IEEE 17th International Symposium on High Performance Computer Architecture (HPCA), pp. 419–430. IEEE (2011)
10. Halpern, M., Zhu, Y., Reddi, V.J.: Mobile CPU's rise to power: quantifying the impact of generational mobile CPU design trends on performance, energy, and user satisfaction. In: Proceedings of 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA), pp. 64–76. IEEE (2016)
11. V8. https://developers.google.com/v8/
12. Tiwari, D., Solihin, Y.: Architectural characterization and similarity analysis of sunspider and Google's V8 Javascript benchmarks. In: Proceedings of 2012 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), pp. 221–232. IEEE (2012)

13. Gavrin, E., Lee, S.J., Ayrapetyan, R., Shitov, A.: Ultra lightweight JavaScript engine for internet of things. In: Companion Proceedings of the 2015 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity, pp. 19–20. ACM (2015)
14. Tilkov, S., Vinoski, S.: Node.js: using javascript to build high-performance network programs. IEEE Internet Comput. **14**(6), 80 (2010)