Journal of information and communication convergence engineering
http://jicce.org

J. Inf. Commun. Converg. Eng. 17(4): 234-238, Dec. 2019 Regular paper

Development of Branch Processing System Using WebAssembly and JavaScript

Moon-Hyuk Choi* **and Il-Young Moon**, ***Member, KIICE***

Department of Computer Engineering, Korea University of Technology and Education, Cheonan 31253, Korea

Abstract

Existing web applications and services have historically been implemented using JavaScript. However, new technologies such as artificial intelligence, the Internet of Things, and Big Data are being developed as part of the Fourth Industrial Revolution. With the definition of the HTML5 web standard, services (such as the technologies mentioned above) that were previously not available through the Web become available. These services, however, need to have the same performance as native applications, and implementing these services will require new technologies. Therefore, additional tools that can work on the Web with native performance are needed. In this paper, a system for branching processing was established using JavaScript and WebAssembly, a language that can operate on the Web. This system performs user requests in advance, and requests are branched in a language that produces faster results. Therefore, a service capable of quick response times can be implemented.

Index Terms: Branch system, JavaScript, WebAssembly, Web performance

I. INTRODUCTION

Web applications that dynamically change content have customarily been implemented in JavaScript, which has historically proven satisfactory for most purposes. Users have been able to use these applications without issue for a wide variety of Internet applications. However, with the Fourth Industrial Revolution, new technologies such as artificial intelligence (AI), the Internet of Things (IoT), and Big Data, in addition to Web standards such as HTML5, have arisen. Various advanced services, such as artificial intelligence and 3D games, are beginning to be offered through the Web.

JavaScript by itself, however, makes it difficult to provide the aforementioned technologies to users, as they require performance nearing that of native applications running on the user's own computer. To solve this problem, a new language or technology, one that can operate on the Web and provide the necessary performance, is required. Towards this goal, the new system should be able to compile low-level code such as C, C++, and RUST. It should also be able to use WebAssembly to provide high performance over the Web [1, 2]. However, WebAssembly is not a wholesale replacement of JavaScript. WebAssembly and JavaScript should be used together. In this paper, a branching system of execution faster than existing systems is proposed. The proposed system operates by branching tasks to the faster operating environment, identified through performance comparisons of WebAssembly and JavaScript [3, 4].

Most web services are implemented by predefining what functions are to be provided to users. As user requests for these services are predefined, expected performance can be calculated in advance using WebAssembly and JavaScript. Based on the above results, the user is provided with service after branching into a runtime corresponding to language that will provide faster performance. Implementing and using a corresponding branching system ensures that users

Received 30 September 2019, **Revised** 06 November 2019, **Accepted** 11 November 2019
***Corresponding Author** Moon Hyuk Choi (E-mail: moon1225@koreatech.ac.kr, Tel:+82-41-560-1493)
Department of Computer Engineering, Korea University of Technology and Education, Cheonan 31253, Korea.

Open Access https://doi.org/10.6109/jicce.2019.17.4.234 print ISSN: 2234-8255 online ISSN: 2234-8883

always get a fast response, regardless of the action requested.

A. Javascript and its Limitations

JavaScript is an object-based scripting language used to implement dynamic behavior on the Web or to process the domain object model (DOM). JavaScript has remained widely used in web programming for over twenty years, because the virtual machines built into most browsers could only support JavaScript. Thus far, JavaScript has been largely sufficient in addressing the technical challenges arising from developing for the Web, and in implementing application functionality without the help of other languages. JavaScript continues to evolve through multiple iterations, such as ECMAScript 6 and 7. As of this writing, the current version is being updated with the latest changes, and its successor is to be named ECMAScript 2020. These developments continue to facilitate implementation on the Web, as JavaScript adds features or functions that can implement new actions whenever a new version is released.

However, even though JavaScript has continued to advance technologically, issues have arisen in terms of speed and performance for delivering the latest technology to users on the web. While there is no problem with the implementation and operation of existing applications in JavaScript, technologies such as AI, the IoT, Big Data, 3D games, AR/ VR, and video processing are starting to be offered through the web. It is difficult to implement applications operating at close to native performance on the Web through JavaScript alone. Therefore, the time has come for other languages to operate on the Web, such as WebAssembly. However, there is currently no language available that can fully replace JavaScript for web services implementation. To solve the above problems, an additional WebAssembly system that can provide near native performance and operate on the web is desirable.

B. WebAssembly

WebAssembly is a language that compiles and delivers code written in low-level languages, such as C, C++, and RUST, as modules that can run on the latest web browsers [5]. The biggest difference between WebAssembly and existing JavaScript implementations is that execution on the Web in WebAssembly can be significantly faster, permitting performance close to that of native applications, because WebAssembly compiles and executes low-level language code. Therefore, the challenge of JavaScript performance limitations can be solved using WebAssembly. In addition, WebAssembly not only provides a performance benefit on the web, but is highly portable. It works on a variety of platforms, as well as on the Web, by operating the features provided by hardware [6, 7].

However, even if a WebAssembly implementation can operate on the Web and on various other platforms, it cannot entirely replace JavaScript [8, 9]. For instance, WebAssembly does not have the ability to access the DOM. Therefore, JavaScript and WebAssembly should be used together to realize the advantages of both. JavaScript does not require compilation because it is a high-level language utilizing dynamic types for implementing web applications. WebAssembly offers fast speeds in a low-level language within a compact binary format. JavaScript and WebAssembly can and should be used simultaneously to synergistically benefit from each other, and thus implement faster and better performing Web services.

II. SYSTEM MODEL AND METHODS

WebAssembly, due to compilation and execution of lower level language code, is faster than JavaScript, and runs at near native execution speed. However, WebAssembly is not necessarily faster than JavaScript for every operation. Therefore, in this section a system is proposed and outlined that can compare JavaScript and WebAssembly performance. Based on this comparison, the system branches into the faster execution environment.

The speed comparison experiment was conducted for three different computations. The first experiment examined matrix multiplication, which is widely used in numerous fields. Because it is also used on the Web for DOM access with JavaScript and CSS styling [10], matrix multiplication makes a suitable test of the new system. The other two experiments examined the factorial and Fibonacci sequences. In these cases, algorithm time-complexity was also frequently used as an example and added as a corresponding experimental element. These three operations are considered representative. Speed comparisons can be performed with other types of operations, but this is not considered critical, because most web services are implemented through a predefined set of functions provided to users. Therefore, it is typically possible to determine the provided services in advance, and to confirm that they will run at improved speed using the branch system.

Matrix multiplication, factorial sequences, and Fibonacci sequences were implemented in C for operation in WebAssembly. Using emscripten, the code implemented in C was converted to a JavaScript module. The module was imported from JavaScript and used [11-13]. After being run in JavaScript, functions were exported, and the module was imported as with WebAssembly, to compare JavaScript and WebAssembly speed within the same page. The operand of each operation was randomized for each trial, to minimize bias in the speed measurements.

III. RESULTS

The computer specifications for comparing JavaScript and WebAssembly speed are listed in Table 1.

The code described in Tables 2-7 was implemented with JavaScript and WebAssembly. Only the key logic sections, not the entirety of the code, are inserted in the tables below. The code tables are listed in the following order: matrix multiplication, factorial sequence, and Fibonacci sequence.

The JavaScript code for implementing matrix multiplication, factorial series, and the Fibonacci sequence, are shown in Tables 2, 3, and 4, respectively.

The WebAssembly code was implemented in C and converted to JavaScript using emscripten. Therefore, C code was inserted, because the converted JavaScript code was excessively long.

The code for implementing matrix multiplication, factorial series, and the Fibonacci sequence with WebAssembly are listed in Tables 5, 6, and 7, respectively. Below is a brief description of the code:

In the matrix multiplication code (Tables 2 and 5), the N = 100,000, and ROW = COL = 3. The code multiplies 3 by 3 matrices. Individual matrix elements are no more than five digits long. After running for 100,000 iterations, the code returns the execution time.

In the factorial code (Tables 3 and 6), N = 10,000. The code executes 10000!, and iterates 10,000 times. Afterwards, the code returns the execution time.

In the Fibonacci code (Tables 4 and 7), N = 45. The code calculates the 45th Fibonacci number. After doing this, the

Table 1. Experimental computer specification

Category	Information
Processor	Inter(R) Core(TM) i5-7200U CPU @ 2.50 GHz 2.7 GHz
Ram	8.00 GB
OS	Windows 10 Pro

Table 2. Matrix multiplication using JavaScript

```
for (let count = 0; count < N; count++) {
  for (let i = 0; i < ROWS; i++) {
    let temp1 = [];
    let temp2 = [];
    for (let j = 0; j < COLS; j++) {
      temp1.push(Math.random() * N);
      temp2.push(Math.random() * N);
    }
    arr1.push(temp1);
    arr2.push(temp2);
  }
}
```

Table 3. Factorial using JavaScript

```
const factorial = num => {
  if (num === 0) return 1;
  return num * factorial(num - 1);
 };
 const N = 10000;
 let start = new Date().getTime();
 for (let count = 0; count < N; count++) {
   factorial(N);
 }
```

Table 4. Fibonacci sequence using JavaScript

```
const fibo = num => {
  if (num === 0) return 0;
  else if (num === 1) return 1;
  else return fibo(num - 1) + fibo(num - 2);
 };
```

Table 5. Matrix multiplication using WebAssembly

```
for(count = 0 ; count < N ; count++) {
    for(i = 0 ; i < ROWS ; i++) {
      for(j = 0 ; j < COLS ; j++) {
        arr1[i][j] = rand() % N;
        arr2[i][j] = rand() % N;
      }
    }
    for(i = 0 ; i < COLS ; i++){
      for(j = 0 ; j < ROWS ; j++) {
        result[j][i] = 0;
        for(k = 0 ; k < ROWS ; k++) {
          result[j][i] += arr1[j][i] * arr2[j][i];
        }
      }
    }
  }
```

Table 6. Factorial using WebAssembly

```
int factorial(int num) {
  if(num == 0) return 1;
  return num * factorial(num - 1);
}
int main(void) {
  for(i = 0 ; i < N ; i++) {
    factorial(N);
  }
  return 0;
}
```

Table 7. Fibonacci sequence using WebAssembly

```
int fibo(int num) {
  if (num == 0) return 0;
  else if (num == 1) return 1;
  else return fibo(num - 1) + fibo(num - 2);
}
int main(void) {
  fibo(N);

  return 0;
}
```

Table 8. Operation speed comparison

-	JavaScript (Sec)	WebAssembly (Sec)
Matrix multiplication	0.2444	0.0262
Factorial computation	3.734067	0.689067
Fibonacci computation	16.09327	19.89853

Table 9. Branch processing speed comparison

Category	Speed (Sec)
Branch Processing	16.563
JavaScript	20.071737
WebAssembly	20.613797

code returns the execution time.

Information is tabulated in Table 8 to facilitate comparison of JavaScript and WebAssembly speed. The experiment was carried out 15 times and the units of results produced were in seconds. The comparison results of JavaScript and WebAssembly computation speed for matrix multiplication, factorial computation, and Fibonacci number computation are listed in Table 8.

It can be confirmed that not all actions in WebAssembly are faster than in JavaScript. It can be seen that the WebAssembly was faster for matrix multiplication and factorial computation, but that JavaScript was faster for Fibonacci sequences.

Based on the results of the experiment, speeds obtained in each environment alone were compared to speeds obtained by branching out the three operations to the faster performing language in each case. For matrix multiplication and factorial computations, WebAssembly performed the execution; JavaScript performed the Fibonacci sequence execution.

The speed comparison between the branch system, the JavaScript-alone system, and the WebAssembly-alone system is listed in Table 9. The JavaScript and WebAssembly results are the sum of the mean values of the previous experiment. The results show that the branching process is faster by approximately four seconds. Based on the experiment, it can be confirmed that the branching system is fastest for completion of a mix of tasks. Thus, applying the branching system to actual Web services execution can enable users to receive service in the fastest way possible, by having any request serviced by the implemented branching processing system.

IV. DISCUSSION AND CONCLUSIONS

JavaScript alone has been sufficient for the implementation of existing applications and web services. However, the Fourth Industrial Revolution and development of the Web mean that applications, services, and technologies that were previously not available on the Web are now being provided. Implementing these through JavaScript alone is difficult because of a requirement of near-native execution speed. Another technology is needed to achieve this. Solutions implemented by compiling and using low-level languages using WebAssembly can operate on the Web at the desired speed.

A system was built to branch execution to the fastest runtime environment available, based on user response time to requests from JavaScript, by comparing JavaScript and WebAssembly execution speed in advance. With this system in place, users obtain the quickest response after submitting the request. In the experiments discussed in this paper, for example, it took 20.071737 s to perform a certain series of operations with JavaScript alone, and 20.613797 s to perform the operation with WebAssembly alone. However, the branching system, by choosing the runtime corresponding to the programming language with faster execution, resulted in a time of 16.563 s, reducing response time by approximately 3.5 s. This is approximately 1.212 times faster than using JavaScript alone, and 1.245 times faster than using only WebAssembly.

In this experiment, a simple mathematical operation was used for testing. If actual performance times are very long, the difference would be magnified, on the order of minutes and hours, not seconds. Thus, a branching system built by comparing JavaScript and WebAssembly performance rates can fulfill users' requests significantly more quickly. As a result, users can receive the fastest responses possible to requests, get better service than provided for by existing services, and experience web services in a more user-friendly manner.

ACKNOWLEDGEMENTS

This research was supported by the Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (No. 2018R1D1A3B07049722).

REFERENCES

[1] InfoWorld, WebAssembly is now ready for browsers to use [Internet], Available: https://www.infoworld.com/article/3176681/webassembly-is-now-ready-for-browsers-to-use.html.

[2] MDN web docs, WebAssembly Concepts [Internet], Available: https://developer.mozilla.org/en-US/docs/WebAssembly/Concepts.

[3] WEBASSEMBLY, [Internet], https://webassembly.org/.

[4] M. Reiser, and L. Blaser, "Accelerate JavaScript applications by cross-compiling to WebAssembly," in *Proceeding of Conference: the 9th ACM SIGPLAN International Workshop*, Vancouver, pp. 10-17, 2017. DOI: 10.1145/3141871.3141873.

[5] Google Developers, Loading WebAssembly modules efficiently [Internet], Available: https://developers.google.com/web/updates/2018/04/loading-wasm.

[6] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien, "Bringing the web up to speed with WebAssembly," in *Proceeding of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, New York: NY, pp. 185-200, 2017. DOI: 10.1145/3062341.3062363.

[7] L. Stephane, O. Yann and D. Fober, "Compiling faust audio DSP code to WebAssembly," in *Proceeding of 3rd Web Audio Conference*, London: UK, 2017.

[8] D. Herrera, H. Chen, and E. Lavoie, "WebAssembly and JavaScript Challenge : Numerical program performance using modern browser technologies and devices," University of McGill, Montreal:QC, Technical report SABLE-TR-2018-2, 2018.

[9] WEBASSEMBLY, Portability [Internet], Available: https://webassembly.org/docs/portability/#asumptions-for-efficient-execution.

[10] MDN web docs, Matrix math for the web [Internet], Available: https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API/Matrix_math_for_the_web.

[11] Google Developers, Emscripting a C library to Wasm [Internet], Available: https://developers.google.com/web/updates/2018/03/emscripting-a-c-library.

[12] A. Zakai, "Emscripten: An LLVM-to-JavaScript compiler" in *Proceeding of Conference: Companion to the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming*, Portland, pp. 301-312, 2011. DOI: 10.1145/2048147.2048224.

[13] A. Zakai, "Fast physics on the web using C++, JavaScript, and emscripten" *Computing in Science & Engineering*, vol. 20, no. 1, pp. 11-19, 2018. DOI: 10.1109/MCSE.2018.110150345.

Moon Hyuk Choi

attends the Department of Computer Engineering at Korea University of Technology and Education. His research interests include the Internet, web standards, and web assembly.

Il Young Moon

received a B.S degree from the Department of Aeronautics and Telecommunications Information Engineering from Korea Aerospace University in 2000, an M.S. degree from the same department of Korea Aerospace University in 2002, and a Ph.D., also from the same department and university, in 2005. His research interests include wireless internet applications, wireless Internet, and mobile IP.