JSOPT: A Framework for Optimization of JavaScript on Web Browsers

MUHAMMAD WAQAS*, AND MINHAJ AHMAD KHAN*

RECEIVED ON 31.10.2016 ACCEPTED ON 21.02.2017

ABSTRACT

In the current era where multi-core technologies are very common in use, the existing web browsers are unable to fully utilize the capability of multi-core processors.The web browsers execute the JavaScript code locally in order to produce an efficient response of web pages. This responsiveness is however limited by the fact that the JavaScript code is uni-threaded, and consequently, the efficiency of the code degrades if it involves a large number of computations.

In this paper, we propose a framework called JSOPT (JavaScript Optimizer) which generates an efficient JavaScript code to effectively utilize multi-core architectures. The framework uses a template containing constructs for communication & synchronization, and subsequently generates optimized code to be executed on the multi-core architectures. Multiple instances of templates are then generated with different implementations of the code and the best instance is selected to be incorporated in the library. With the optimized code generated using JSOPT, our results show a significant improvement in the performance of several benchmarks involving intensive computations based matrix operations on the Mozilla Firefox web browser.

Key Words: JavaScript, Threads, Performance, Web Browsers, Code Generation.

1. INTRODUCTION

JavaScript is the most popular scripting language and is supported by almost all of the modern web browsers. It is used to leverage the websites by allowing dead web pages to interact with users, thereby improving the response time and interactivity. In contrast to Java, the JavaScript code is embedded inside HTML (Hyper Text Markup Language) [1]. The web browsers are required to run the HTML applications, thereby constraining the execution of the JavaScript code inside a single thread created for their execution [2].

In recent years, the hardware technology has evolved as well from a single processing unit to multiple processing units within a system. The shared memory systems, in this regard, contain tightly coupled processing units so that the memory can be shared among these units. Despite these advancements, the JavaScript code is far away from exploiting the parallelism and gaining performance improvement with multiple processing units [3]. This limitation arises from the fact that the execution of the JavaScript code is totally browser dependent which often

Corresponding Author (E-Mail: vickycs50@gmail.com)
* Department of Computer Science, BahauddinZakariya University, Multan.

becomes unresponsive. Most of the web browsers get into an such a state because of two basic reasons:

- Heavy Computations: The web browser becomes unresponsive if the code contains very heavy computations in its loop iterations or it contains iterations where the termination condition is not easily met.
- Intensive DOM Activities: The code may perform heavy manipulation of objects in the DOM (Document Object Model), and consequently, the performance degrades to such a state in which the browser becomes unresponsive.

The entire workload is transferred to a single processing unit whereas the rest of the processing units remain idle. Consequently, the performance efficiency of the code deteriorates. Unfortunately, most of the web browsers and the applications executing on them are not set to exploit the multi-core architectures despite the usage of heavy JavaScript code.

Although the functions such as **setTimeout** and **setInterval** [4] may be used in JavaScript in an asynchronous manner to simulate a parallel execution, the execution of JavaScript still remains inside a single thread. Multi-threaded execution of JavaScript, in contrast, means dividing the script into small segments and treating each segment as an independent thread. Each thread has its own resources as well apart from the UI (User Interface) thread. This requirement led to the development of Web Worker API [5-6] which is newly introduced in HTML5 [7]. The Web Worker API (Application Program Interface) is used to boost the power of JavaScript. It helps the programmers to write their JavaScript code in small segments and execute every segment as an independent thread simultaneously. Moreover, it also makes the script utilize all the available resources without putting any extra load on the UI thread

In this paper, we aim at performance improvement of the JavaScript code being executed on a shared memory system. We propose a framework called JSOPT that takes architectural specification as input and generates an optimized code that is able to exploit the parallelism due to the existence of multiple processing units. While generating optimized code, the framework makes use of a template containing basic constructs of communication and synchronization. The framework instantiates the template with different code versions, searches for the best instance of the code and generates an optimized version of the code in the form of a JavaScript library.As benchmarks, we use problems with complex mathematical computations such as finding the matrix of minors and cofactors, matrix power, matrix multiplication and matrix inverse [8], which inherently execute extremely slow in JavaScript. Our results show that the code produced by the proposed framework significantly improves the performance of these benchmarks.

For generating optimized library similar to the one produced by JSOPT, some high performance libraries such as FFTW (Fastest Fourier Transform in the West) [9] and SPIRAL[10] also make use of an iterative approach. These libraries however target the C++ based implementations where the threaded support is provided natively or through other libraries such as OpenMP and MPI. In contrast to these approaches, the JSOPT framework optimizes code for JavaScript which requires the use of special Web Worker API. There have been efforts for improving the performance of JavaScript code [11-13] or estimating the quality of JavaScript Code as described in [14], however, to the best of our knowledge, this is the first attempt to improvethe performance of the JavaScript code on multi-core architectures in an automated way.

The rest of the paper is organized as follows. In Section 2, we discuss the multithreaded execution context of JavaScript and the related work. Section 3 describes the architecture of

the proposed JSOPT framework. The main features of the JSOPT framework are succinctly discussed in Section 4. The experimental setup and performance results are presented in Section 5. The conclusion and future work are discussed in Section 6.

2. CONTEXT AND RELATED WORK

Over the past few years, the use of multi-cores in a single system has increased remarkably. Similarly, a huge growth has been seen in the use of internet applications. To ensure an efficient execution of the JavaScript code, the programmers expect the web browsers to perform necessary activities required for an efficient execution of code.

JavaScript has a syntax almost similar to Java, however, it makes use of a model based architecture which is implemented by almost all the modern browsers. The cross-browser portability of JavaScript code is degraded due to non-availability of an appropriate JavaScript Framework. To cope with this issue, a prototype model to measure existing JavaScript frameworks is proposed by Graziotin et.al. [15]. The model makes use of empirical feedback from practitioners regarding metrics for a JavaScript framework. This research work however results in better selection of a framework instead of generating automated code as provided by the JSOPT framework.

Another model aimed at improving the JavaScript code uses the metrics based on size, complexity and maintainability [16]. Their research work targets the evaluation of different existing JavaScript frameworks which may be utilized by programmers to write JavaScript code in their applications. The metrics used in this study however do not target the execution speed as targeted by the JSOPT framework proposed in this paper.

A thread-level speculation based mechanism implemented in the JavaScript Engine is shown to improve performance of web applications [17]. The proposed strategy allocates each iteration of a loop to a separate thread similar to the conventional threading mechanism. In case of violation of dependencies, the execution is rolledback to a previous point. The smaller granularity of parallelism results in a small improvement in performance of the code in comparison with the our suggested approach.

The research study in [18] discusses top free JavaScript frameworks in terms of their main characteristics (e.g. reactiveness, scalability and models and features. Most of the frameworks discussed in the study provide specialized features while aiming at the ease of development of the JavaScript code. These frameworks do not target the optimization of code for multi-core architectures, in contrast to the our suggested approach.

A programming model for a distributed environment is presented by Welc et.al. [19]. The suggested model permits an application to be executed on multiple clients, while exploiting the resources in a proper way. The proposed model incorporates two major modules, for local and remote execution, respectively. The Netscape Plugin API is used to provide native support for the local module, whereas, the remote component deploys the Web Worker API [5-6] running on top of the Google's V8 JavaScript execution engine [20].

Richards et. al. [21] analyzed the usage of various elements for better performance of the programs written in JavaScript. The programs in the JavaScript language use features which are different from those of the static typed languages whose code may be optimized in different ways. Any programs not exploiting the dynamic features of the JavaScript code may be optimized using the conventional approaches after static analysis. Similarly, the research work by Ratanaworabhan et. al. [22] analyzes the behavior of various benchmark suites (such as SunSpider) and commercial websites. The analysis reveals that the

benchmarks are not proper representatives of the commercial websites, thereby emphasizing on different parameters of performance than those used in the benchmark suites. Similarly, execution behavior of the social networks like facebook is analyzed by Martinsen et.al. [23] through a deterministic execution of use cases. The research work in these strategies mainly aims at analysis of the performance factors or characteristics in a single-threaded environment. Our approach, in contrast, targets performance improvement in a multi-threaded environment while using responsiveness as the main parameter.

3. JSOPT FRAMEWORK ARCHITECTURE

The proposed framework JSOPT is implemented as a prototype and can be used to generate optimized JavaScript code for mathematical problems in the form of a library. Currently, the mathematical problems involving the widely used matrix operations have been optimized. Using optimized code, efficient results are produced without blocking the web browser's UI thread.

The main architecture of the JSOPT framework is depicted in Fig. 1. It makes use of a template comprising three sub-templates (**Main**, **onMessage**, **CollectData**), corresponding to a routine for which the optimized JavaScript code is to be generated. As shown in Fig. 1, the architecture specification is initially determined and used as input by the code generator. The template contains slots that are filled with n different versions of (a library routine/function) code $C_{1,2,...,n}$. A version of code is placed in the template to generate the function code which is then executed and profiled to generate timing reports. The code version with the minimum execution time is then added to the optimized library code.

An example format of the **Main** sub-template code used by the code generator is given in **Algorithm-1.** Let $\mathbf{X_0}$ be the number of cores, and **P** be the size of input. The value **X** representing the slice of input which will be passed to a unique thread, is computed at step 3. Let **GResult** be the resultant matrix and $\mathbf{T_c}$ be the number of threads having completed execution. The **GResult** and $\mathbf{T_c}$ are the global variables and will be used by the **CollectData** function to be described later. The loop at step 6 is executed for all the cores. The code in $\mathbf{Code_{init}}$ is invoked to perform initialization activities, such as computing the starting and ending indices of the input data. Let the set **A** represent input arguments to be passed to the **postMessage** function which in turn invokes the **onMessage** function. Let **S** and **E** represent respectively the sets for starting and ending indices corresponding to the input arguments. The data is sent to each thread by invoking the **postMessage** function and passing it as arguments the sets **A**, **S** and **E** at step 11. After having performed computations, the final result is required to be accumulated in a global array **GResult** by using the function **CollectData** for each thread. Its call is set to be queued at step 12 for later execution.

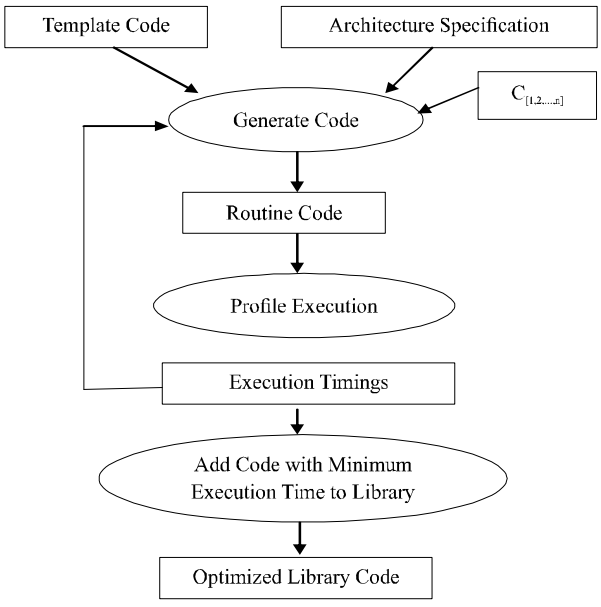

FIG. 1. JSOPT WORKING MECHANISM

Algorithm-2 shows the steps taken by the function **onMessage** for executing the code whenever a message containing input data and starting/ending indices is received by a thread. Initially, a 2-dim array **R** is declared to store output result corresponding to an individual thread. Let $\mathbf{u}_1$ and $\mathbf{v}_1$ be the start indices and $\mathbf{u}_2$ and $\mathbf{v}_2$ be the ending indices for the output matrix **R** in which the result is stored. Subsequently, The $\mathbf{Code}_{uni}$ is invoked at step 4 to execute the uni-threaded code to store result in corresponding indices. It exists as an inline macro call and is able to use the globally declared variables. The matrix **R** with indices $\mathbf{u}_1$, $\mathbf{v}_1$, $\mathbf{u}_2$, and $\mathbf{v}_2$ is then passed to the UI thread for results accumulation via a call to **postMessage** function at step 5.

The code written in the **onMessage** sub-template is implemented in a separate .js file and is executed in parallel for each thread. The message sent by each thread in the **postMessage** function call then results in invocation of the **CollectData** function.

Algorithm-3 describes pseudo-code for the **CollectData** sub-template, whose call was set to be queued in Algorithm 1. This function receives results (the array **R** and the indices $\mathbf{u}_1$, $\mathbf{v}_1$, $\mathbf{u}_2$, $\mathbf{v}_2$) from each thread and stores it in corresponding indices of the global array **GResult** in steps 3 to 7. It also increments the thread count variable $\mathbf{T}_c$, and invokes the $\mathbf{Code}_{finalize}$ function for finalization activities, when all the threads have completed execution. Using this function, the results of independent computations are accumulated in the resultant array **GResult**.

```
1.  Begin
2.      // Let P be the input size and X0 be the
        number of cores
3.      Set X = P/X0
4.      // Let GResult and Tc be the global resultant
        matrix and thread variable, respectively
5.      SetTc = 0
6.      For J = 1 To X0
7.          // Initial code, compute indices
8.          CallCodeinit
9.          //Let A, S, and E represent the sets of
            arrays, starting indices and ending indices,
            respectively
10.         // Send Data To Threads
11.         CallpostMessage (A, S, E)
12.         SetonMessage = CollectData
13.     End For
14. End
```

ALGORITHM-1. MAIN SUB-TEMPLATE CODE

```
1.  Begin
2.      // Let R be the resultant matrix
3.      // Let u1&v1 be the start indices and u2 &v2
        // be the ending indices for the result R
4.      CallCodeuni (X, S, E, R, u1, v1, u2, v2)
5.      CallpostMessage (R,u1, v1, u2, v2)
6.  End
```

ALGORITHM-2. ONMESSAGE SUB-TEMPLATE CODE

4. FEATURES OF THE LIBRARY GENERATED BY THE JSOPT FRAMEWORK

All the execution of code is required to be performed on the web browser. On a multi-core system, the JSOPT framework adopts a mechanism in which the Web Worker API support [3,5] is exploited.

4.1 Portability

Using JSOPT framework, if the Web Worker API support exists on a web browser, the optimized library functions

```
1.  Begin
2.  // Let u1&v1, be the start indices and u2&v2
        // be the ending indices for the result GResult
3.  For I = u1To v1
4.      For J = u2To v2
5.          SetGResult [I,J] = R [I,J]
6.      End For
7.  End For
8.  SetTc = Tc + 1
9.  IfTc = X Then
10.     // Finalization Code
11.     Callcodefinalize
12. Endif
13. End
```

ALGORITHM-3. COLLECTDATA SUB-TEMPLATE CODE

generate multiple threads, otherwise all the work is done inside a single thread. This characteristic is referred to as feature detection and is used to provide portability for different browsers. Although there are a few feature detection libraries [24-25] but none of them targets performance improvement in order to provide portability of executing code on different browsers in an efficient manner.

4.2 Time Efficiency

The optimized library is generated by iteratively searching for the best version of code in terms of the time efficiency. The improvement in performance is obtained by executing individual function code on the cores actually existing in the system.

4.3 Scalability

As the number of cores grows on the client machine, more efficient results are produced by executing the optimized code generated by the JSOPT framework.

5. EXPERIMENTATION SETUP AND RESULTS

Web workers don't share memory so all the messages are copied to be passed between threads, because of this nature we use Transferable Objects [26-27] where the ownership of filled buffer is passed to worker thread. We use Mozilla Firefox for our experimentation which is considered to be the most widely used open source web browser, supported by most of the platforms ranging from modern smartphones to the high-end servers.

We made use of several benchmarks including the computation of matrix of minors and cofactors, matrix power, matrix multiplication and matrix inverse. The experimentation is performed on a machine having Intel Core i5 processor with 2 physical cores, 4 logical cores and 4 GB of RAM (Random-Access Memory) while running Windows-8 operating system. For every benchmark, the performance is computed in terms of execution time in seconds. To represent performance improvement, we use the metric of speedup which is the ratio of the time taken by the original sequential code to the time taken by the optimized code.

5.1 Matrix of Minors and Cofactors

The performance results obtained for calculating the matrix of minors and cofactors are shown in Fig. 2. The X-axis contains the order of the input square matrix for which the computation is performed.

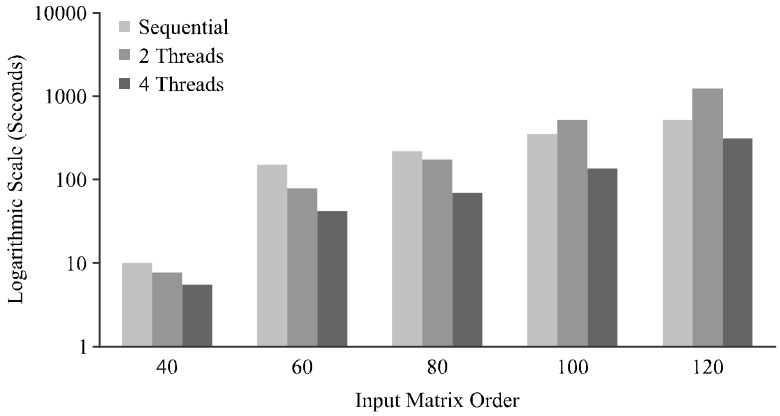

FIG. 2. TIME ANALYSIS GRAPH FOR MATRIX OF MINORS AND COFACTORS

The parallel code for 2 threads as produced by the JSOPT framework performs better than the sequential code. It attains an average speedup of 1.11, with the highest speedup of 1.89. Similarly, using 4 threads as well, the parallel optimized code generated by the JSOPT framework performs better than the sequential code. It attains an average speedup of 2.56, with the highest speedup of 3.53. It is also evident from the results that the performance of the optimized code for calculating the matrix of minors and cofactors increases with the increase in the number of cores.

5.2 Matrix Power

The results to calculate the square of an input matrix using matrix power code are shown in Fig. 3. The order of the input matrix is given on X-axis.

The parallel code for 2 threads as produced by the JSOPT framework performs better than the sequential code. It attains an average speedup of 1.08, with the highest speedup of 1.40. Similarly, using 4 threads as well, the parallel optimized code generated by the JSOPT framework performs better than the sequential code. It attains an average speedup of 1.49, with the highest speedup of 2.80. The results also show that the performance of the optimized code for calculating the matrix power increases with the increase in the number of cores.

5.3 Matrix Multiplication

The performance results obtained for calculating the matrix multiplication of two square matrices are shown in Fig. 4. The X-axis contains the order of the input square matrices for which the matrix multiplication is performed.

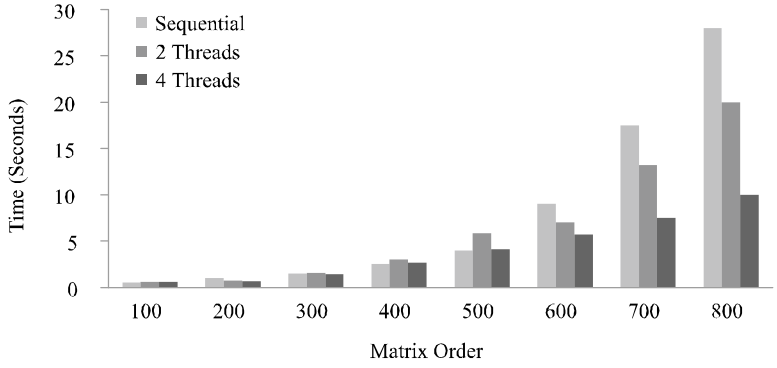

FIG. 3. TIME ANALYSIS GRAPH FOR MATRIX POWER

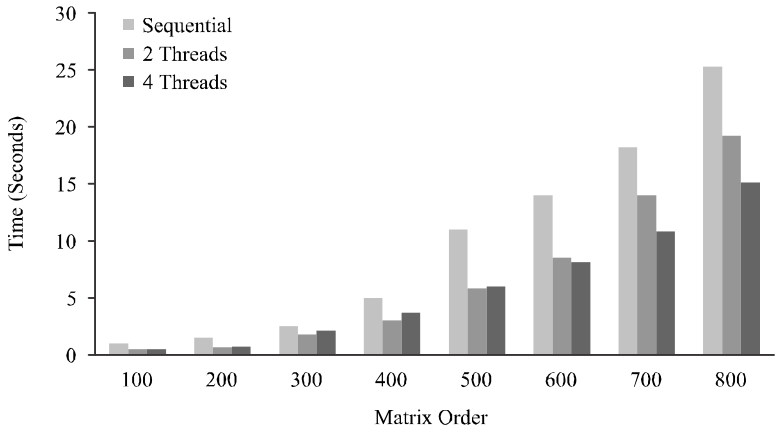

FIG. 4. TIME ANALYSIS GRAPH FOR MATRIX MULTIPLICATION

Similar to the matrix power results, the parallel code for 2 threads as produced by the JSOPT framework for matrix multiplication also performs better than the sequential code. It attains an average speedup of 1.69, with the highest speedup of 2.30. Similarly, using 4 threads as well, the parallel optimized code generated by the JSOPT framework performs better than the sequential code. It attains an average speedup of 1.70, with the highest speedup of 2.14.

5.4 Matrix Inverse

The performance results obtained for computing the matrix inverse are shown in Fig. 5. The X-axis contains the order of the input square matrix for which the inverse is computed.

Using 2 threads, the optimized code attains an average speedup of 0.85 which implies that the optimized code does not perform better than the sequential code for a small number of threads. Using 4 threads, however, the parallel optimized code generated by the JSOPT framework performs better than the sequential code. It attains an average speedup of 2.66, with the highest speedup of 3.36.

6. CONCLUSION

In the last few years, a huge increase has been seen in the use of internet applications. For better performance of these applications, it is required to offload complex applications into small segments and distribute workload into multiple threads to take full advantage of available resources such as multiple cores.

In this paper, we have proposed a framework called JSOPT that is able to produce optimized JavaScript code for a system with multiple cores. The framework uses the architectural specification and generates optimized JavaScript code while invoking the Web Worker API for parallel execution. A template containing constructs for communication & synchronization is instantiated with multiple versions of a function code. The code is then profiled to find the best version to be incorporated in the optimized library. The optimized code generated using JSOPT is able to achieve average speedups of 1.11, 1.08, and 1.69 using 2 cores for computation of the matrix of minors and cofactors, matrix power, and matrix multiplication, respectively. Similarly, using 4 cores, the optimized code is able to produce average speedups of 2.56, 1.49, 1.70 and 2.66 for computation of matrix of minors and cofactors, matrix power, matrix multiplication and matrix inverse, respectively.

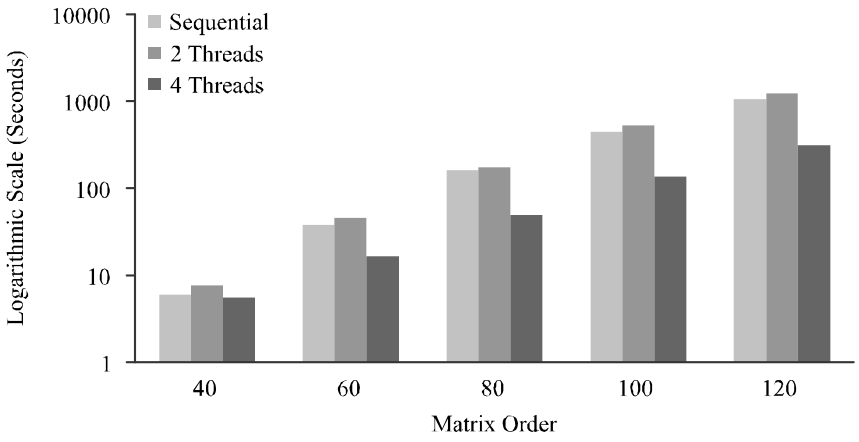

FIG. 5. TIME ANALYSIS GRAPH FOR MATRIX INVERSE METHOD

The Web Worker API deployed in our suggested approach is limited to support only the message passing mechanism and also lacks any suitable synchronization constructs. As future work, we intend to enhance the Web Worker API by developing a hierarchical thread spawning mechanism with the hierarchy starting from a parent thread different from the UI thread together with implementation of other matrix operations such as matrix determinant and transpose.

ACKNOWLEDGEMENT

The authors are thankful to the Department of Computer Science, Bahauddin Zakariya University, Multan, Pakistan, for providing facilities required to carry out this research work as part of the M.Sc. Thesis.

REFERENCES

[1] Quigley, E., "JavaScript by Example", 2nd Edition", Prentice Hall, New Jersey, 2010.

[2] Rousset, D., "Introduction to HTML5 Web Workers: The JavaScript Multi-threading Approach", Microsoft Development Network, Available at:http://msdn.microsoft.com/en-us/hh549259.aspx.

[3] Bidelman, E., "The Basics of Web Workers", HTML5 Rocks, Available at:http://www.html5rocks.com/en/tutorials/workers/basics.

[4] Mozila, "JavaScript timers", Mozilla Developer Network, 2015. [Online]. Available: https://developer.mozilla.org/en-US/Add-ons/Code_snippets/Timers.

[5] Hickson, I., "Web Workers W3C Working Draft", Available at: https://www.w3.org/TR/workers/.

[6] Green, I., "Web Workers And Big Data–A Real World Example", Available at: http://greenido.wordpress.com/2012/05/20/Web-workers-and-big-data-a-real-world-example.

[7] Clark, R., Studholme, O., Murphy, C., and Manian, D., "Beginning HTML5 and CSS3", Apress Publishers, New York, 2012.

[8] Bronson, R., "Schaum'sOutline of Matrix Operations", Schaum Outline Series, New York, 2011.

[9] Frigo, M., and Johnson, S., "The Design and Implementation of FFTW3", IEEE Proceedings, Volume 93, No. 2, pp. 216-231, 2005.

[10] Puschel, M., Moura, J., Singer, B., Xiong, J., Johnson, J., Padua, D., Veloso, M., and Johnson, R., "SPIRAL: A Generator for Platform-Adapted Libraries of Signal Processing Algorithms", International Journal of High Performance Computing Applications, Volume 18, No. 1, pp. 21-45, 2004.

[11] Souders, S., "Even Faster Websites—Performance Best Practices for Web Developers", O'Reilly Media, USA, 2009.

[12] Osmani, A., "Writing Fast, Memory-Efficient JavaScript", Smashing Magazine, Available at: http://www.smashingmagazine.com/2012/11/05/writing-fast-memory-efficient-javascript/.

[13] Souders, S., "High Performance Websites-Essential Knowledge for Front-End Engineers", O'Reilly Media, USA, 2007.

[14] Misra, S., and Cafer, F., "Estimating Quality of JavaScript", International Arab Journal of Information Technology, Volume 9, No. 6, 2012.

[15] Graziotin, D., and Abrahamsson, P., "Making Sense Out of a Jungle of JavaScript Frameworks - Towards a Practitioner Friendly Comparative Analysis", Proceedings of 14th International Conference on Product-Focused Software Development and Process Improvement, Lecture Notes in Computer Science, pp. 334-337, Springer-Verlag, 2013.

[16] Gizas, A., Christodoulou, S., and Papatheodorou, T.,"Comparative Evaluation of JavascriptFrameworks", Proceedings of 21st International Conference Companion on World Wide Web Companion, 2012.

[17] Martinsen, J., Grahn, H., and Isberg, A., "Using Speculation to Enhance JavascriptPerformance in Web Applications", IEEE Internet Computing, Volume 17, No. 2, pp. 10-19, March, 2013.

[18] Alex, I., "Top 23 Best Free JavaScript Frameworks for Web Developers 2016", Tools, Colorlib, 2016. [Online]. Available: https://colorlib.com/wp/javascript-frameworks/.

[19] Welc, A., Hudson, R., Shpeisman, T., and Adl-Tabatabai, A., "Generic Workers", Programming Support Innovations for Emerging Distributed Applications on PSI EtA, 2010.

[20] Google, "Chrome V8 | Google Developers", Google Developers. [Online]. Available: https://developers.google.com/v8/.

[21] Richards, G., Lebresne, S., Burg, B., and Vitek, J., "An Analysis of the Dynamic behavior of JavaScript Programs", ACM SIGPLAN Notices, Volume 45, No. 6, pp. 1, May, 2010.

[22] Ratanaworabhan,P., Livshits, B., and Zorn, B., "JSMeter: Comparing the behavior of JavaScript Benchmarks with Real Web Applications", Proceedings of USENIX Conference on Web Application Development, Berkeley, USA, 2010.

[23] Martinsen, J., and Grahn, H., "A Methodology for Evaluating JavaScript Execution behavior in Interactive Web Applications", 9th IEEE/ACS International Conference on Computer Systems and Applications, 2011.

[24] Watson, A., "Learning ModernizrCreate Forward-Compatible Websites using Feature Detection Features of Modernizr", Birmingham, Packt Publishing, 2012.

[25] Corti, S., "HTML5 Browser and Feature Detection", MSDN Magazine, Microsoft Inc., Available at: http://msdn.microsoft.com/en-us/magazine/hh475813.aspx.

[26] Bidelman, E., "Transferable Objects: Lightning Fast!", HTML5 Rocks, Available at: http://updates.html5rocks.com/2011/12/Transferable-Objects-Lightning-Fast.

[27] Mozilla, "ArrayBuffer", Mozilla Developer Network. [Online]. Available: https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/ArrayBuffer.