International Journal of Multimedia and Ubiquitous Engineering
Vol.9, No.7 (2014), pp.171-184
http://dx.doi.org/10.14257/ijmue.2014.9.7.15

A Parallelization Design of JavaScript Execution Engine

Duan Hucai[1,2], Ni Hong[2], Deng Feng[2] and Hu Linlin[2]

[1]*National Network New Media Engineering Research Center, Institute of Acoustics, Chinese Academy of Sciences, Beijing 10190, China*
[2]*University of Chinese Academy of Sciences, Beijing 100049, China*
duanhc@dsp.ac.cn

Abstract

With more and more consumer electronics apply multi-core chips, the traditional serialized JavaScript execution engine, which is optimized by just-in-time (JIT) compilation technology, fails to utilize multi-core advantages. This paper proposes a mathematical model to detect the dependency of serial JavaScript tasks and a parallelism execution algorithm for serial JavaScript execution engines. Moreover, the parallel JavaScript execution engine with thread-level speculation technology is implemented based on the SquirrelFish Extreme engine of WebKit. As the experiment were conducted respectively on the general test platform platform Sunspider in the industry and world top 15 websites at traffic volume, the results indicate that both in the real Web application and Sunspider platform, the parallel JavaScript execution engines with 2 to 16 threads can raise the performance dramatically compared with a SquirrelFish execution engine with or without JIT acceleration, respectively.

Keywords: *Multi-core chips, parallel computation, thread-level speculation, JavaScript*

1. Introduction

JavaScript is a dynamic interpretation and execution language oriented to objects. As JavaScript acts as main scripting language in the client's Web application, its execution performance directly affects the user's experience related to Web applications. The traditional solution of JavaScript acceleration is JIT compilation technology [1-4] which generate machine codes from running JavaScript codes for execution. JIT acceleration technology may increase the compilation expenses [4], provided that the optimized code is executed again and the type of JavaScript objects is not changed *etc*. JIT optimization technology promotes JavaScript performances conventionally through experiments on the specific test platform [5, 6], but the actual features of Web applications are different from the test platform of industrial circles [79]. JIT optimization technology usually fails to reduce the JavaScript execution time in common Web applications according to some researches [10]. Moreover, as more and more consumer electronics use multi-core chips, JavaScript execution engine applies serialization operation with JIT acceleration technology and fails to take advantage of multi-core chips. According to some research results [11], the JavaScript execution engine under serialization operation is parallelized so that the performance of Web applications is raised by 45 times. To reduce the difficulty of parallel compilation, it is proposed in [12-14] that the technology of Thread-Level Speculation (TLS) [15] should be applied to parallelize JavaScript engine and realize JavaScript function-level parallelism. However, the parallelism at function level cannot explore the parallelism potential of JavaScript application to the maximum extent

[1] Project supported by the National Science and Technology Support Program of China (No.2012BAH73F01) and CAS pilot special issue (No. XDA06040501).

ISSN: 1975-0080 IJMUE

while a large number of loop operations in the actual JavaScript application can apply fine-grained loop level to partition parallelism.

This paper analyzes the factors for JavaScript function-level or loop-level parallelism, proposes the detection approach of data dependency in JavaScript byte codes and further discusses an algorithm which divide serialized JavaScript program into tasks for parallel execution. Meanwhile, based on SquirrelFish Extreme, it also proposes a parallelism solution of byte-code interpreters. The algorithm hereof improves performances obviously after being tested on world top 10 websites at traffic volume.

2. Feasible Analysis of Parallel JavaScript Execution Engine

2.1. Principle Analysis of JavaScript Execution Engine

The conventional JavaScript execution engine applies serial execution and translates JavaScript codes into byte codes for interpretation and execution [1, 3] or directly into machine codes for execution [2]. Figure 1 shows the execution processes for three common types of JavaScript execution engines.

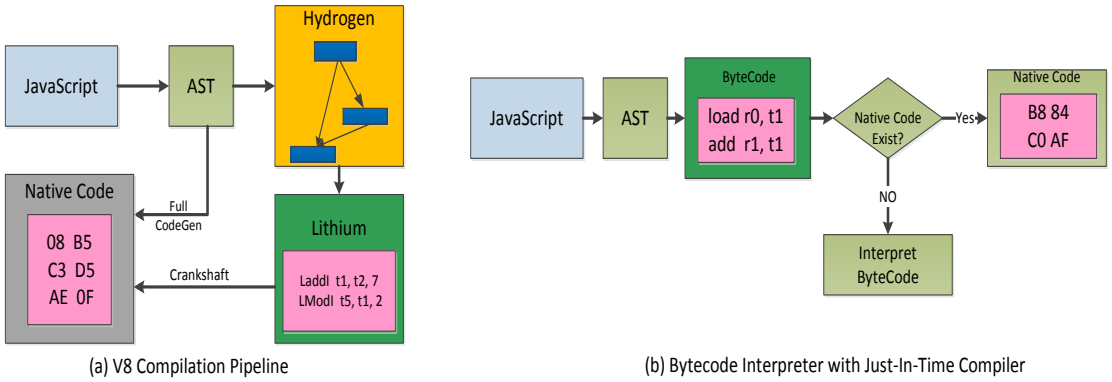

Figure 1. Execution Processes of JavaScript Execution Engines

As shown in Figure 1, it indicates the flow diagrams about how mainstream JavaScript execution engines interpret and execute JavaScript programs at present. (a) indicates V8's execution process, V8 converts JavaScript source codes into abstract syntax trees and initiates a Full-Codege compiler to compile the abstract syntax trees into machine codes related to the platform and execute them one by one; it initiates a Crankshaft compiler to optimize hotspot functions in the running process and generate better machine codes for execution. (b) is a common JavaScript execution engine with the combination of a byte code interpreter and a JIT compiler, where JavaScript source codes are converted into abstract syntax trees. Different from V8, this engine generate a kind of abstract byte codes while an interpreter interprets and executes these codes. Meanwhile, a JIT compiler compiles hotpots of byte codes into machine codes related to the platform to optimize performances. As both SquirrelFish Extreme and TraceMonkey apply this design, the difference lies in optimized grain sizes. And the former applies the hotspot optimization algorithm based on Method while the latter uses that based on Trace [16]. Therefore, the current JavaScript execution engine can execute JavaScript codes only according to the single-threaded sequence but fails to speed up with the advantages of multi-core chips.

1.2. Parallelism Instances of JavaScript Execution Engine

All parts of the JavaScript program always have a certain dependency relationship. The parallel processing shall decompose the program into several tasks for parallel execution without causing damage to these dependency relationships in order to shorten the time required for running the whole process. However, there is no general model for the parallelism of a serial program at present, so it is necessary to take the features of serial tasks in consideration for partition. As there are a lot of functions in the JavaScript program, it is possible to perform parallel execution of callee and caller functions during the function call. Moreover, there are a lot of loop iterative operations in the JavaScript codes and each iterative operation may be also executed in a parallel manner. Figure 2 indicates an example of creating JavaScript function, while JavaScript source codes stay on the left and the corresponding byte-code pseudo-code on the right. In this figure, the sequence of byte codes is partitioned into 13 tasks. Obviously, Tasks 2 and 3 are mutually independent and so are Tasks 3 and 4. The execution must be carried out after Task 1. And Tasks 4-12 are the byte-code sequence after JavaScript loop unrolls and every round of iteration is based on the former round. The loop iterative operation cannot be executed in a parallel manner while Task 13 cannot be executed until all other tasks are completed.

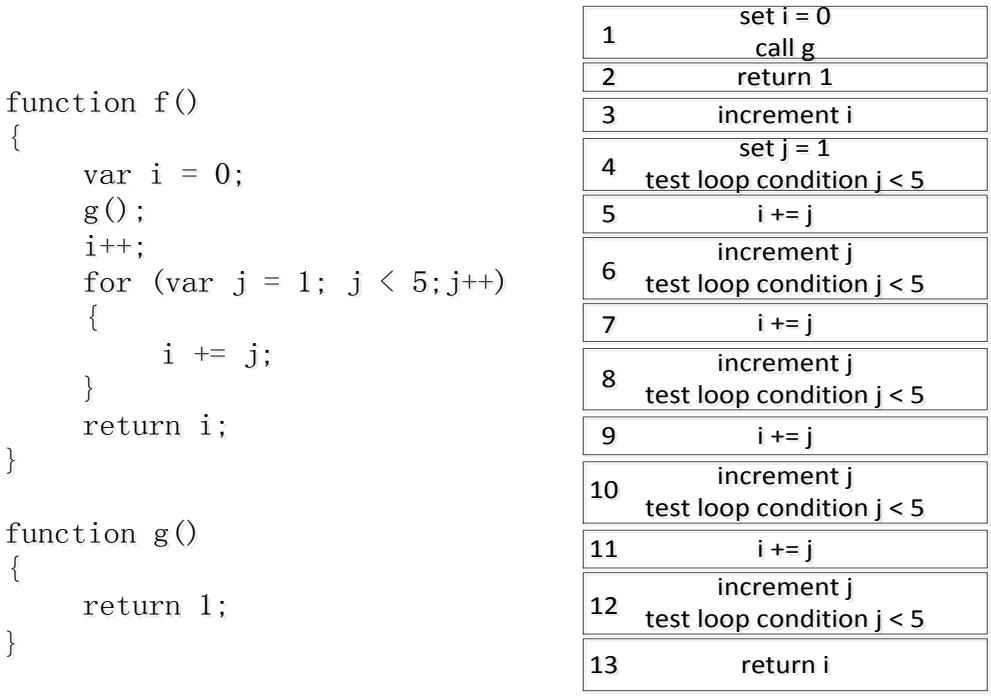

Figure 2. Example of JavaScript Source Code and its Corresponding Byte-Code Sequence

Figure 3 indicates the sequence chart of the parallelism execution for the tasks in Figure 2. First, execute Task 1 in the main thread. When it is time to execute Task 2, open a thread to execute Task 2 because Tasks 2 and 3 are mutually independent. As the main thread proceeds to other byte codes after Task 2, it is time to execute Task 3. And then, open another thread to execute Task 3 due to mutual independence between Tasks 3 and 4. As the main thread proceeds to other byte codes after Task 3, kick off Task 4. The execution of Task 5 relies on

Tasks 3 and 4, so Task 4 must be executed in series by the main thread at this time. After the completion of Task 4, the main thread must not execute Task 5 until the completion of Task 3. As Task 3 is completed, it sends feedback to the main thread and the main thread executes Task 5 and its subsequent tasks in order. Therefore, comparing with serial execution, parallel execution can reduce the total time indicated in Equation (1):

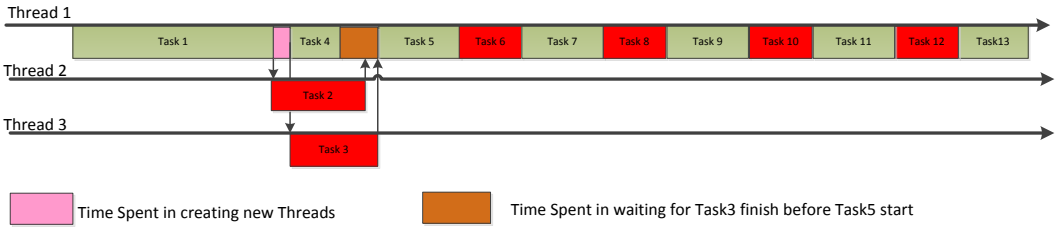

Figure 3. Sequence Chart of JavaScript Parallel Execution

$$T_{t2} + T_{t3} - T_{thread} - T_{wait5} \quad (1)$$

Where, T_{t2} refers to the time consumed for Task 2 execution, T_{t3} indicates the time consumed for Task 3 execution, T_{thread} means the time consumed for opening Threads 2 and 3 while T_{wait5} indicates the time consumed to wait for the completion of Task 3 before the main thread executes Task 5, it can be expressed in Figure 3 as follows:

$$T_{wait5} = T_{t3} - T_{t4} \quad (2)$$

Therefore, the time reduced for parallel execution can be expressed as follows:

$$T_{t2} + T_{t4} - T_{thread} \quad (3)$$

Because Tasks 2 and 4 are executed in a parallel manner, the time consumed for the two tasks is reduced, comparing with serial execution but additional time is consumed for opening the thread. Provided the time consumed for serial execution of Tasks 2 and 4 is longer than that for opening the thread, the parallelization can improve the performances of JavaScript program. In the actual program, the expenses for task execution is much more than the time consumed for opening the thread, so maximizing the parallel execution of JavaScript program can improve performances and it is feasible to parallelize the serial JavaScript program.

2. Parallel algorithm of JavaScript Execution Engine

The key to parallel execution of JavaScript execution engine is to identify the dependency relationship in JavaScript tasks. Under the premise of no damage to the dependency relationship, a JavaScript program is partitioned into several subtasks for parallel execution. The dependency relationships can be categorized into the two types of control dependency relationship and data dependency relationship while the former leads to program process changes and the latter is incurred by reading/writing the same data. And the data dependency can affect the dependency relationship of a JavaScript program. Therefore, the analysis theory and techniques of data dependency relationships is the basis of JavaScript program parallelism.

2.1. Mathematical Model Analysis of JavaScript Task Dependency

$P_i(i = 1, 2, \cdots, m)$ is defined as one task in the JavaScript program and $V_i(i = 1, 2, \cdots, n)$ as a variable. As different tasks carry out reading/writing operations on the same variable, there

may be data dependency relationships. $m \times n$-order matrix M_r is defined as reading matrix with the value as follows:

$$M_r(i, j) = \begin{cases} 0, \text{ no reading Variable } V_j \text{ in } P_i, \\ 1, \text{ reading Variable } V_j \text{ in } P_i. \end{cases} \quad (4)$$

$m \times n$-order writing matrix is defined as:

$$M_w(i, j) = \begin{cases} 0, \text{ no writing Variable } V_j \text{ in } P_i, \\ 1, \text{ writing Variable } V_j \text{ in } P_i. \end{cases} \quad (5)$$

There are three types of data dependency relationships: read-after-write (RAW), write-after-read (WAR) and write-after-write (WAW) [17]. $m \times m$-order matrix M_{wr} of JavaScript task read-write dependency is defined as

$$M_{wr} = M_r \times M_w^T \quad (6)$$

Therefore:

$$M_{wr}(i, j) = \begin{cases} 0, \text{ no data RAW dependency between } P_i \text{ and } P_j \\ k(0 < k \le n), \text{ k variables have RAW dependency between } P_i \text{ and } P_j. \end{cases} \quad (7)$$

$m \times n$-order matrix M_{ww} of JavaScript task WAW dependency is defined as

$$M_{ww} = M_w \times M_w^T \quad (8)$$

Therefore,

$$M_{ww}(i, j) = \begin{cases} 0, \text{ no data WAW dependency between } P_i \text{ and } P_j, \\ k(0 < k \le n), \text{ WAW dependency between } P_i \text{ and } P_j. \end{cases} \quad (9)$$

$m \times n$-order dependency matrix M_d is defined as:

$$M_d = M_{wr} + M_{ww} \quad (10)$$

And

$$M_d(i, j) = \begin{cases} 0, \text{ no dependency between} P_i \text{and } P_j, \\ k'(0<k' \le 2n), \text{ dependency between } P_i \text{and } P_j. \end{cases} \quad (11)$$

Meanwhile, the dependency relationship is of transitivity; if $M_d(i, j) \neq 0$ and $M_d(j, l) \neq 0$, there is the dependency relationship between P_i and P_l. The task can be executed in a parallel manner only if there is no dependency relationship while the tasks under dependency relationships must be executed by the same thread in order.

2.2. Parallel algorithm of JavaScript Execution Engine

C is defined as a set of tasks under execution or to be executed and $C \subseteq \{P_1, P_2, P_3, \cdots, P_m\}$; every time a new task $P_i (i = 1, 2, \cdots, m)$ is kicked off, put the new task is put into Set C; and when a task is completed, delete it from C. $C_j \in C, 0 < j \le m$ is defined as some task under execution, $Thread(C_j)$ is the thread where some task is under execution at that time, $P(Thread(C_j))$ indicates the task set which has

dependency relationship with C_j and is to be executed in the same thread with C_j, and $P(Thread(C_j)) \subseteq C$. $Index(P_i)$ refers to the serial numbers of Task P_i in $\{P_1, P_2, P_3, \cdots, P_m\}$ while $F \in \{P_1, P_2, \cdots, P_m\}$ is a task to be executed. This paper proposes an algorithm for parallel execution of serial JavaScript tasks:

Step 1: If $C = \varnothing$, execute F in the main thread and put F into Set C, otherwise proceed to Step 2;

Step 2: If $M_d(Index(F), Index(C_j)) = 0$ to random $C_j \in C$, execute F in the new thread and put F into Set C, otherwise proceed to Step 3;

Step 3: If $M_d(Index(F), Index(C_j)) > 0$ to $C_j \in C$, put F in C and $P(Thread(C_j))$, proceed to Step 4;

Step 4: As the execution of C_j is completed, delete C_j from C; if $P(Thread(C_j)) \neq \varnothing$, $Thread(C_j)$ goes on proceeding to the next $C_{j'}, C_{j'} \subseteq P(Thread(C_j))$ from $P(Thread(C_j))$ and . Meanwhile, delete $C_{j'}$ from $P(Thread(C_j))$, otherwise complete $Thread(C_j)$ and feed the results back to the main thread. And then proceed to Step 5;

Step 5: The main thread skips F and analyzes the subsequent tasks, proceed to Step 1 if there is any task, otherwise wait for execution completion of all threads and return to execution results.

3. Algorithm Implementation

This paper applies thread-level speculation technology and proposes a parallel algorithm solution of the JavaScript execution engine based on the Squirrelfish Extreme execution engine.

3.1. Algorithm for Serial JavaScript Task Partitioning

For parallel execution of the JavaScript program, it is required to divide the serial program into several tasks and find tasks from the task set for parallel execution. The thread-level speculation is intended to dynamically parallelize the serial program while there are the parallelism grain-sizes at loop level and method level. The loop-level parallelism assigns one thread to iteration of each loop for execution while the method-level parallelism deploys each function as one thread. Squirrelfish Extreme execution engine first translates JavaScript source codes into byte codes [18], partitions serial JavaScript program into several tasks in the process of byte-code generation and establishes the task dependency matrix. “Loop”, “loop_if_true” and “loop_if_less” of Squirrelfish Extreme byte codes initiate one-loop instructions, its “target” parameter is a start address of one loop-level task, the “call” instruction is a function call instruction, its “func” parameter is a start address of the function-level task and the “ret” instruction is the symbol indicating calling by a function and backing to results.

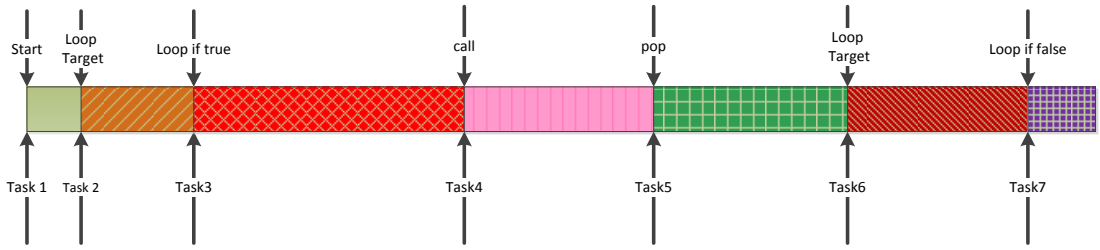

Figure 4. Schematic Diagram of Task Partitioning for JavaScript Serial Program

Algorithm 1 Tasks Division for Serial JavaScript

```
Initialize task id as 1, establish tasks hashmap and add task 1 and opcode address 0 into it;
For each opcode do
   For each argument in opcode do
      If argument is a loop||loop_if_true||loop_if_less then
            task id increment;
            add task id and this opcode target argument into the tasks hashmap;
        Else if argument is a call then
            task id increment;
             add task id and this opcode func argument into the tasks hashmap;
        Else if argument is a ret then
             task id increment;
             add task id and this opcode address into the tasks hashmap;
        Else
              If last argument is a loop_if_true||loop_if_less then
                   task id increment;
                   add task id and this opcode address into the tasks hashmap;
              End if
          End if
     End for
End for
```

Figure 5. Pseudo-Code Realization of Task Partitioning for JavaScript Serial Program

Figure 4 is the schematic diagram of task partitioning for the JavaScript serial program, it scans the byte-code sequence in order; every time it meets with a call function instruction or a loop iterative instruction, record one new task and map the task number to the address parameter of the instruction. Meanwhile, byte-code fragments between iteration and function call are partitioned into separate tasks. Figure 5 shows the pseudo-codes of the solution.

3.2. Dependency Matrix Computation

Figure 6 shows the pseudo codes realized by the algorithm; as JavaScript program is partitioned into several tasks, it is possible to obtain $M_r(i,:)$ from the i[th] row of the reading matrix and $M_w(i,:)$ from the i^{th} row of the writing matrix through all read/write variables in the speculation task $P_i(i=1,2,\cdots,m)$. The same operations are carried out for all tasks to establish the reading matrix M_r and the writing matrix M_w and obtain the task dependency matrix M_d in combination with Equations (6)-(11). To reduce the memory consumption, use a sparse matrix during design to store the data of M_r, M_w and M_d.

Algorithm 2 Interdependencies matrix calculation

```
Initialize elements of sparse matrix Md as 0;
Initialize elements of sparse matrix Mr and Mw as 0;
Initialize variable tables;
For each task do
    for each opcode do
        for each argument in opcode do
            if argument is a write then
                get the task id and the variable id, mark Mw(task id, variable id) = 1;
            else if argument is a read then
                get the task id and the variable id, mark Mr(task id, variable id) = 1;
            end if
        end for
    end for
end for
get Md from Mw and Mr;
```

Figure 6. Pseudo-Code Computations through JavaScript Task Dependency Matrix

3.3. Parallel Execution of Serial JavaScript

After the dependency matrix is obtained, it is possible to judge whether two tasks can be executed in a parallel manner depending on the matrix-to-matrix dependency value. Execute the tasks with dependency relationships in the same thread and apply thread pool technology during realization to reduce the expenses for creating threads, while the pseudo-code realization is shown in Figure 7.


```
Algorithm 3 Parallel execution for Serial JavaScript
{Main Thread execution}
Initialize thread pool T;
Initialize queue Wtask for tasks executing or waiting for execution;
For each task do
   Add the task id into the Wtask;
   If Wtask is empty do
        Execute the task in the main thread;
   Else if Md(task id, id of any task in Wtask) = 0 do
        Get a new thread from thread pool T;
        Execute the task in the new thread;
   Else if Md(task id, id of any task in Wtask) = 1 do
        Add the task id into the dependency task thread's task queue;
   End if
   If the task is complete do
        Remove the task from Wtask;
   End if
   If there is the next task exist do
        Process the next task;
   Else
        Waiting for others threads completing and return the result;
   End if
End for

{Child Thread execution}
Initialize queue Stask for tasks waiting for this thread execution;
For each task do
   Execute the current task;
   If the task is completion do
        Remove this task from Wtask;
        If Stask is not empty do
            Get the first task in Stask;
            Execute the task;
        Else
             Return the result to the main thread;
             Gave back the thread to the thread pool T;
        End if
   End if
End for
```

Figure 7. Pseudo Codes for JavaScript Parallel Execution

4. Performance Evaluation

This paper conducted the performance evaluations respectively from the two aspects of theory and experiment and fully approved that the parallelized JavaScript execution engine can significantly improve the performances of JavaScript program.

4.1. Theoretical Analysis

Suppose that the serial JavaScript program can be partitioned into several tasks, the execution time of each task $P_i(i=1,2,\cdots,k)$ is $T_i(i=1,2,\cdots,k)$ respectively. Comparing with the execution time of each task, the expenses are omitted for allocating threads because the thread pool is used to allocate threads. Therefore, the JavaScript execution engine without JIT acceleration requires the time $\sum_{i=1}^{k} T_i$ to execute k tasks.

Suppose that the serial JavaScript program is finally executed in parallel by d threads and each thread executes $k_j(j=1,2,\cdots,d)$ tasks, the JavaScript execution engine in the serialized design requires the time $\max(\sum_{i=1}^{k_j} T_i)$ to execute k tasks.

For JavaScript execution engine with JIT acceleration, suppose that the execution time for byte codes accelerated accounts for $\alpha(0<\alpha<1)$ ×100% of the total tasks, the time consumed for optimizing the part of codes accounts for $\beta(0<\beta)$ ×100% of the total task execution time and the execution efficiency of JIT codes compiled is $f(0<f)$ times better than that of the original byte codes, so the JavaScript execution engine with JIT acceleration requires the time $(1-\alpha+\beta+\frac{\alpha}{f})\times\sum_{i=1}^{k} T_i$ to execute k tasks.

Therefore, comparing with the conventional serial JavaScript execution engine without JIT acceleration, the parallel execution engine can improve the performances by:

$$\frac{\sum_{i=1}^{k} T_i}{\max(\sum_{i=1}^{k_j} T_i)}(j=1,2,\cdots,d) \tag{12}$$

Obviously, it is proved from the theoretical view, the parallel JavaScript execution engine can effectively improve the performances by d times at the maximum only if the serial JavaScript program can be partitioned into separate tasks. Meanwhile, the d value is related to the specific JavaScript program which can be proved by the subsequent tests.

Comparing with the serial JavaScript execution engine with JIT acceleration, the parallel execution engine hereof can improve the performances by:

$$\frac{(1-\alpha+\beta+\frac{\alpha}{f})\times\sum_{i=1}^{k} T_i}{\max(\sum_{i=1}^{k_j} T_i)}(j=1,2,\cdots,d) \tag{13}$$

According to Pareto's 80/20 Law, 20% of the codes in a program take up 80% of the total running time, so the value of α is set to 80% [19] in this paper. And the average execution time for SquirrelFish Extreme byte codes is 10 times higher than that for the machine codes [12, 18], so f is set to 10 and β to 20%. Equation (13) is converted into:

$$\frac{0.48\times\sum_{i=1}^{k}T_i}{\max(\sum_{i=1}^{k_j}T_i)}(j=1,2,\cdots,d) \tag{4}$$

The parallel execution efficiency can significantly exceed that of JIT acceleration only if the serial JavaScript program can be executed in parallel at the maximum. During value settings of Equation (13), the value of Parameter β depends on the seeking efficiency of hotspots and its subsequent execution frequency of its JavaScript program hotspots while the actual value of β is absolutely possible to exceed 1 [20].

No general theories can be used at present to carry out quantitative analysis for improving the parallelization performances of serial JavaScript program, so it is required to perform experimental statistics for the common Web applications in order to analyze the performance improvement due to parallel JavaScript execution engine.

4.2. Experimental Analysis

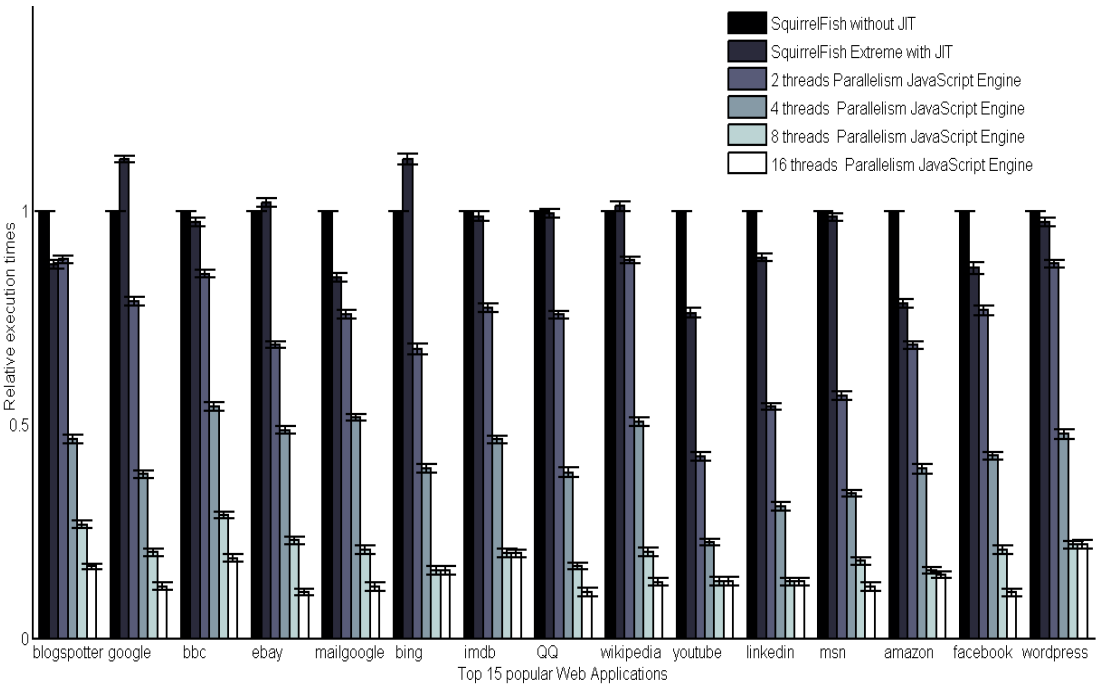

Figure 8. The Comparative Chart of Test Performances on World Top 15 Websites at Traffic Volume

The experiment in this paper was based on the embedded network set-top box and the tests were conducted on the general test platform Sunspider of 15 websites with different business types as well as industrial fields. Figure 8 shows the comparative chart of performance tests on world top 15 websites with maximum traffic volumes, where JIT acceleration indicates unobvious effects on the actual websites, the average performances are raised by 5.43% and JIT acceleration even reduces the performances on some websites. Parallel JavaScript execution engine can significantly improve performances; comparing with serial JavaScript

execution engine without JIT acceleration, the parallel JavaScript engines with 2, 4, 8 and 16 threads can improve the average performances respectively by 37.07%, 1.36 times, 4.08 times and 5.92 times; comparing with the serial JavaScript execution engine with JIT acceleration, the parallel JavaScript execution engines with 2, 4, 8 and 16 threads can improve the average performances respectively by 30.01%, 1.24 times, 3.82 times and 5.57 times.

Figure 9 shows the test results on Sunspider platforms. The characteristics of JavaScript codes better fit the expectation of JIT optimization on Sunspider platforms, so it is usually used as performance test platform for JIT acceleration.

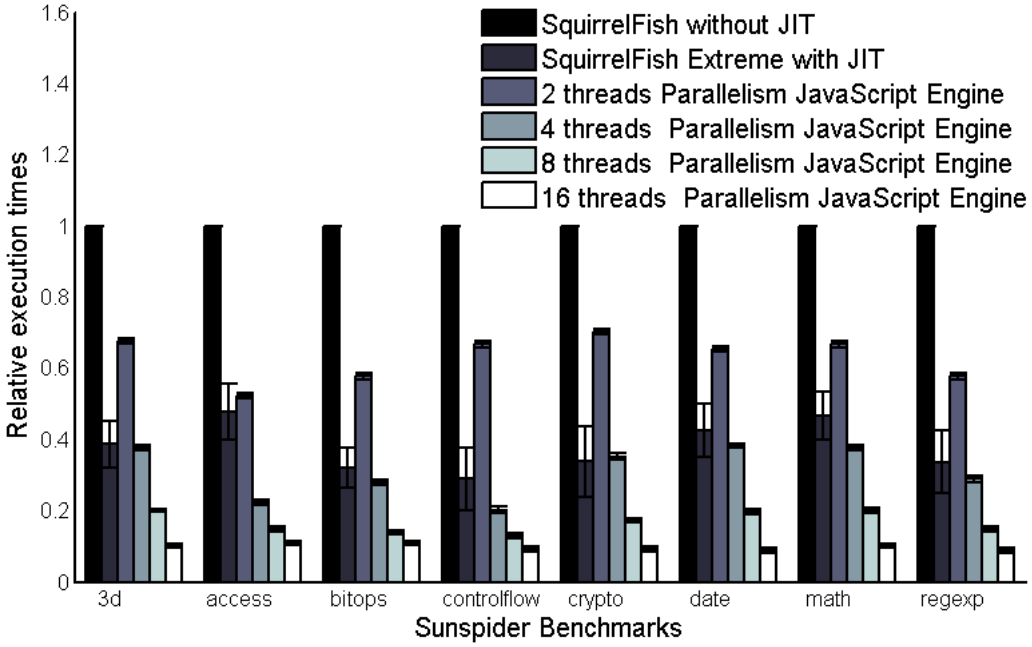

Figure 9. the Comparative Chart of Performance Tests on Sunspider Platforms

On Sunspider platforms, comparing to the JavaScript execution engine without JIT acceleration, JIT acceleration can improve the average performances by 1.62 times but Sunspider's features are obviously different from the actual Web business. Comparing to the JavaScript execution engine without JIT acceleration, the parallel JavaScript execution engines with 2, 4, 8 and 16 threads can raise the average performances respectively by 58.57%, 2.23 times, 4.97 times and 9.28 times; and comparing with JavaScript serial execution engine with JIT acceleration, the parallel JavaScript execution engines with 2, 4, 8 and 16 threads can raise the average performances respectively by -39.7%, 22.86%, 1.27 times and 2.91 times.

Obviously, the characteristics of Sunspider are different from that of current mainstream Web applications and the parallel JavaScript execution engines with the same degree of parallelism have different effects on performance promotion for two types of businesses. For mainstream Web applications, 16-thread parallelism and 8-thread parallelism both have unobvious performance improvement, but the application of 16 threads on Sunspider nearly doubles performances comparing with that of 8 threads. Therefore, more-thread parallelism is not always better while it requires combining JavaScript task features and identifying the maximum degree of parallelism under current parallelism grain size through experiments.

5. Conclusions

As the serial JavaScript execution engine fails to utilize multi-core advantages at present, this paper proposes a design method of the parallel JavaScript execution engine. The test results from actual Web applications and Sunspider platforms indicate that in the actual Web applications, comparing with a serial JavaScript execution engine without JIT acceleration, the parallel JavaScript execution engines with 2, 4, 8 and 16 threads can raise the performances respectively by 37.07%, 1.36 times, 4.08 times and 5.92 times; comparing with a serial JavaScript execution engine with JIT acceleration, the parallel JavaScript execution engines with 2, 4, 8 and 16 threads can raise the performances respectively by 30.01%, 1.24 times, 3.82 times and 5.57 times. On a Sunspider platform, comparing with a serial JavaScript execution engine without JIT acceleration, the parallel JavaScript execution engines with 2, 4, 8 and 16 threads can raise the performances respectively by 58.57%, 2.23 times, 4.97 times and 9.28 times, while comparing with a serial JavaScript execution engine with JIT acceleration, the parallel JavaScript execution engines with 2, 4, 8 and 16 threads can raise the performances respectively by -39.7%, 22.86%, 1.27 times and 2.91 times.

This algorithm utilizes the parallelism of function-level and loop-level grain sizes at the same time during the execution and significantly improves the performances of Web business with a lot of loops or function calls, while current Web businesses also contain many non-loop or –function-call codes. It is also possible to execute these codes in parallel, so the subsequent study may highlight the parallelism research of byte-code grain sizes. Meanwhile, the parallel JavaScript execution engine may generate additional memory consumption. Therefore, the memory optimization will also be the key for future researches.

Acknowledgements

Thanks to my tutor, from the topics of papers, writing, changes to the draft, condensed tutor's efforts and wisdom, thank the fund support.

References

[1] G. Garen, Announcing SquirrelFish.2008.http://www.webkit.org/blog/189/announcing-squirrelfish/.
[2] Google Inc. A New Crankshaft for V8.2010. http://blog.chromium.org/ 2010/12/new-crankshaft-for-v8.html.
[3] MozillaWiki. JavaScript:TraceMonkey. 2010. https://wiki.mozilla.org/ JavaScript:TraceMonkey.
[4] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat and M. Franz, "Trace-based just-in-time type specialization for dynamic languages", In ACM Sigplan Notices, vol. 44, no. 6, (**2009**), June, pp. 465-478, ACM.
[5] WebKit. SunSpider 1.0.2 JavaScript Benchmark. 2013. https://www.webkit.org/perf/sunspider-1.0.2/sunspider-1.0.2/driver.html.
[6] Google. V8 Benchmark Suite.2009. http://v8.googlecode.com/svn/data/benchmarks/v3/ run.html.
[7] J. K. Martinsen and H. Grahn, "A methodology for evaluating JavaScript execution behavior in interactive web applications", In Proc. of the 9th ACS/IEEE Int'l Conf. On Computer Systems and Applications, (**2011**), December, pp. 241–248.
[8] P. Ratanaworabhan, B. Livshits and B. G. Zorn, "JS Meter: Comparing the behavior of JavaScript benchmarks with real web applications", In WebApps'10: Proc. of the 2010 USENIX Conf. on Web Application Development, (**2010**), pp. 3–3.
[9] G. Richards, S. Lebresne, B. Burg and J. Vitek, "An analysis of the dynamic behavior of JavaScript programs", In PLDI '10: Proc. of the 2010 ACM SIGPLAN Conf. on Programming Language Design and Implementation, (**2010**), pp. 1–12.
[10] Martinsen, J. Kasper, H. Grahn and Anders Isberg, "A comparative evaluation of JavaScript execution behavior", Web Engineering. Springer Berlin Heidelberg, (**2011**), pp. 399-402.
[11] E. Fortuna, "A limit study of JavaScript parallelism", Workload Characterization (IISWC), 2010 IEEE International Symposium on. IEEE, (**2010**).

[12] Martinsen, J. Kasper, H. Grahn and A. Isberg, "Using speculation to enhance javascript performance in web applications", Internet Computing, IEEE 17.2 (**2013**), pp. 10-19.
[13] Martinsen, J. Kasper and H. Grahn, "An alternative optimization technique for JavaScript engines", Proceedings of the Third Swedish Workshop on Multi-Core Computing (MCC-10), (**2010**).
[14] Martinsen, J. Kasper and H. Grahn, "Thread-level speculation for web applications", Second Swedish Workshop on Multi-Core Computing, (**2009**).
[15] Oancea, E. Cosmin, and M. Alan, "Software thread-level speculation: an optimistic library implementation", Proceedings of the 1st international workshop on Multicore software engineering, ACM, (**2008**).
[16] A. Gal, "Trace-based just-in-time type specialization for dynamic languages", ACM Sigplan Notices, vol. 44, no. 6, ACM, (**2009**).
[17] S. Gupta, "SPARK: A high-level synthesis framework for applying parallelizing compiler transformations", VLSI Design, 2003, Proceedings, 16th International Conference on. IEEE, (**2003**).
[18] WebKit, Squirrelfish bytecode, 2013. http://www.webkit.org/specs/squirrelfish-bytecode.html.
[19] Martinsen, J. Kasper, H. Grahn and A. Isberg, "Preliminary Results of Combining Thread-Level Speculation and Just-in-Time Compilation in Google's V8", Sixth Swedish Workshop on Multicore Computing (MCC-13), Halmstad University, (**2013**).
[20] Martinsen, J. Kasper, H. Grahn and A. Isberg, "A comparative evaluation of JavaScript execution behavior", Web Engineering, Springer Berlin Heidelberg, (**2011**), pp. 399-402.

