Automatic Parallelism in JavaScript

Rosetta Reatherford
Computer Science and Software Engineering
Rose-Hulman Institute of Technology
Terre Haute, Indiana
rosetta.reatherford@gmail.com

Dr. Micah Taylor
Computer Science and Software Engineering
Rose-Hulman Institute of Technology
Terre Haute, Indiana
taylormt@rose-hulman.edu

***Abstract*—Automatic parallelism is a new frontier in computer science. There has been some success in various languages of achieving automatic parallelism. Largely, these tools have been for C and C++. JavaScript, one of the most widely used programming languages on the internet, has had few attempts at automatic parallelism. These attempts have been held back by issues with client-side differences and vague scopes inherent to JavaScript as a language. However, we have successfully managed to convert JavaScript to parallel code by using a compiler which enforces ECMAScript style on coders. This avoids some of the issues inherent to analyzing JavaScript code for possible parallelization. The code is able to be compiled by developers and then uploaded on their servers. This ensures that any users with multiple cores are able to execute the parallel sections. The focus of this paper was on well-defined for-loops.**

***Index Terms*—Parallelism, for-loops, JavaScript, closure, compiler**

I. Introduction

Parallelism is an issue being increasingly addressed in technology and programming. Sadly, Moore's law has become more of a curse than a gift, which is driving multicore development [1]. However, for one of the world's leading languages on the internet, JavaScript, attempts at parallelism have not been able to robustly address making the language more parallel. Despite introducing Web Workers in 2009, most programmers do not take advantage of the JavaScript supported threads. This is largely credited to the many issues Web Workers have for programmers. New solutions are on the horizon which would aide development, although there are libraries which seek to enhance JavaScript threading currently. This thesis explores using a JavaScript compiler to identify parallel JavaScript for well-defined for loops.

A. Motivation

JavaScript is one of the most popular languages on the web. As of 2011, it 95% of internet users had a browser capable of handling JavaScript [2]. The capabilities of the language make it attractive for programmers to use in order to make rich and interactive web pages. However, there are limitations to what the language can do as it relies heavily on single-threaded client-side execution. Even though HTML5 introduced Web Workers which allow multithreaded application of JavaScript, many programmers have shied from using them in their code [3]. This slow-down can be attributed to difficulties with concurrency as well as possible issues with the complex and sometimes confusing ways in which Web Workers access and pass information between themselves [4].

Parallelism is a well-defined issue with a few solutions. While the nature of JavaScript can differ significantly from other languages and requires a different approach [5], there has been research conducted on how to optimally parallelize code, such as through thread-level speculation which dies upon dependency violations [6].

B. Use of the Closure Compiler

The flexible and easy-to-used nature of Google's Closure Compiler makes it attractive to use. Typically, it is used as a minifying and optimization tool that reduces unnecessary or redundant code as well as unneeded white space to speed download time for clients. The way the program works also makes it quite attractive for static analysis tools. It converts JavaScript to Java through abstract syntax trees and then back to JavaScript after modification. The process of Closure Compiler is detailed in a later section.

The ability of the compiler helped manage some of the issues with JavaScript usually encountered when trying to parallelize or analyze the code. This ability sets our approach aside from others. It only requires developers to run the code through the compiler and upload the ParallelJS library [7]. There is no extra work needed on the part of the client in order to begin taking advantage of the parallel code. The process also allows developers to have more control. They can review the changes before committing to them.

C. Use of ParallelJS

```
var a;
var print;
var x;

for(var i = 0; i < 10; i++) {
                a[i] = i;
}
```

Figure 1: The Code converted to an AST in Figure 2.

The use of the library, ParallelJS, is purely an issue of reducing complexity. It is a lightweight library that eases the use of parallel Web Worker code within JavaScript. It allows running functions within the current file as parallel. With other Web Worker code, the functions that a developer is attempting to run in parallel over certain data must be placed into separate files. The lightweight nature of the library made it attractive for the purposes of this paper.

II. Closure Compiler

Google's Closure Compiler is an open source compiler for JavaScript. The compiler converts JavaScript into Java through abstract syntax trees and back to JavaScript after modification. The process of the Closure Compiler makes it unique in its interpretation of JavaScript data. Even without the explicit type annotations, the analysis is usually able to easily infer what is intended by the developer. The break down into the abstract syntax trees allows them to be modified and altered in a fashion that allows for the trees to be checked for any problems which could cause errors.

A. Overview of Compilation

Compilation is done in a few steps. This is given by an overview within Google's documentation [8]. The first step is to create the Compiler instance and process the command line arguments given by the user. It then parses the code into an Abstract-Syntax Tree (AST). Next, the compiler runs all the Compiler passes that will make modifications to the AST. The AST Tree is then converted to JavaScript and output to a file. There are other optional commands which can change this process, but this is the typical use case.

B. The Abstract-Syntax Tree (AST)

The AST that the code is converted to has a great deal of information attached to each node [9]. This information can be accessed through simple commands that allow a developer to tell the type of the node and receive context for its role. In example is shown in Figure 2 and Figure 1. This shows the complicated way the compiler breaks the code into an AST for conversion into JavaScript.

C. Modification

Modification from add-on developers, such as the method, approached by our paper, is done mainly through Compiler Passes [10]. A Compiler Pass modifies the AST by traversing the nodes within the trees. Nodes can be added, removed, or modified in order to make changes to the tree.

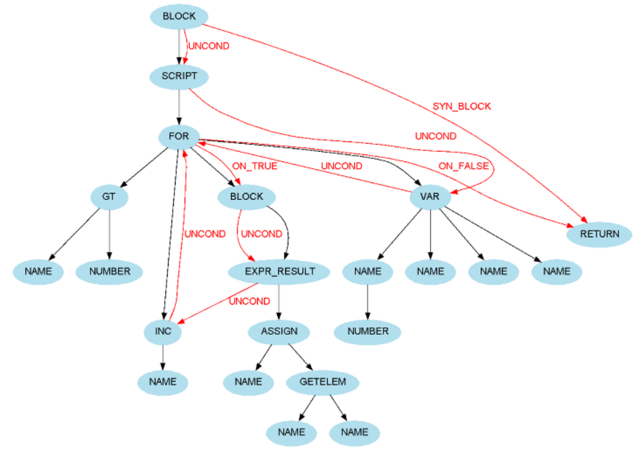


Figure 2: The AST for the code in Figure 1.

The advantage of this method is that not much understanding is needed of the Closure Compiler while allowing a great degree of information and avoiding large modification to the overall process of the compiler. This allowed us to access a lot of information about the code without needing to parse and interpret it ourselves. This saves time and allows for the focus to be on interpreting the nodes for parallelism than attempting to interpret the scope and possible issues that occur with static analysis of JavaScript.

III. Approach

The approach taken was to focus efforts on for-loops to narrow the scope of variables and restraints for parallelizing blocks of code within these areas. The simple dependencies considered are below, erring on the side of caution and avoidance of dependency issues.

A. Dependency rules

1) A variable may not be equal to itself unless it is in an array

Variables may be equal to any expression as long as that expression is not the same variable. For example,

$$x = x + 1;$$

is not a valid parallelizable expression. However, this is not true for the case of arrays, where the value may be imported through the JSON data. For example,

$$a[i] = a[i] + 1;$$

is a valid parallelizable expression.

2) An array variable may not contain different math expressions to access the array at a certain point

An array variable may contain an expression in order to access a value at the result of that expression. However, this must be a consistent expression for it to be treated as possibly parallelizable. For example,

$$a[i + 1] = a[i + 1] + 1;$$

is a valid expression, while

$$a[i + 1] = a[i + 2] + 1;$$

is not a valid expression.

B. Analysis Process

```
var numberofcores = navigator.hardwareConcurrency || 4, Parallel, a = [], x, r, c = 3, i = 0,
    originalLimit = 100, additionalToiteratorii = Math.ceil((originalLimit - i) / numberofcores);
for (var iteratorii = 0;iteratorii < numberofcores;iteratorii++) {
  var data = Array(4);
  data[0] = iteratorii * additionalToiteratorii + i;
  data[1] = originalLimit < additionalToiteratorii + data[0] ? originalLimit : additionalToiteratorii
    + data[0];
  data[2] = [];
  var paralleldata = new Parallel(data);
  paralleldata.spawn(functionfoo0).then(functionfoo0callback);
}
function functionfoo0callback(b) {
  x = b[3];
  for (var d = b[0];d < b[1];d++) {
    a[d + 1] = b[2][d];
  }
}
function functionfoo0(b) {
  for (var d = b[0];d < b[1];d++) {
    b[2][d] = d + b[3] + b[3] + b[3] + b[3], b[3] = b[2][d] + 1;
  }
  return b;
}
;
```

Figure 4: Result of the code in Figure 3.

The analysis done within the Compiler Pass is done by a process of node traversal. First, the compiler pass does a post-order traversal through the AST to skim for for-loops. When a for-loop is found, it begins traversal over the internal code block of the for-loop. This traversal finds variable assignments and saves the left-hand side in a set, then traverses through the right-hand side for any violation of the dependency rules listed above. If a violation is found, a flag is set that notifies the Compiler Pass that the code block is not able to be parallelized. This results in the code being ignored and remaining unconverted to the parallel format. When all variables assigned and used are found, the Compiler Pass begins the process of converting the for-loop into parallel code.

The conversion begins by dividing the total amount of work over the number of cores the client computer has. This is a variable number which is common to JavaScript. Once this is done, the Compiler Pass creates an array to store JSON data that will be sent to the worker function. It then calculates where the thread should begin and end, saving those in the newly created array. Any values which the thread will need to know for expressions will also be placed into the array. The Compiler Pass utilizes the ParallelJS library to create and spawn a thread which will call the worker function and assigns the callback to the callback function. A new for loop is created which iterates over the number of threads in order to do this for each thread that will be spawned. This newly created for-loop is added to a list that will replace the original for-loop after this traversal has finished.

The next step is redoing the original for-loop. The Compiler Pass then begins the process of renaming variables

```
var Parallel;
var a = Array();
var x, r;
var c = 3;


 for(var i = 0; i < 100; i++) {
     a[i + 1] = i + x + x + x + x;
     x = a[i+1] + 1;
}
```

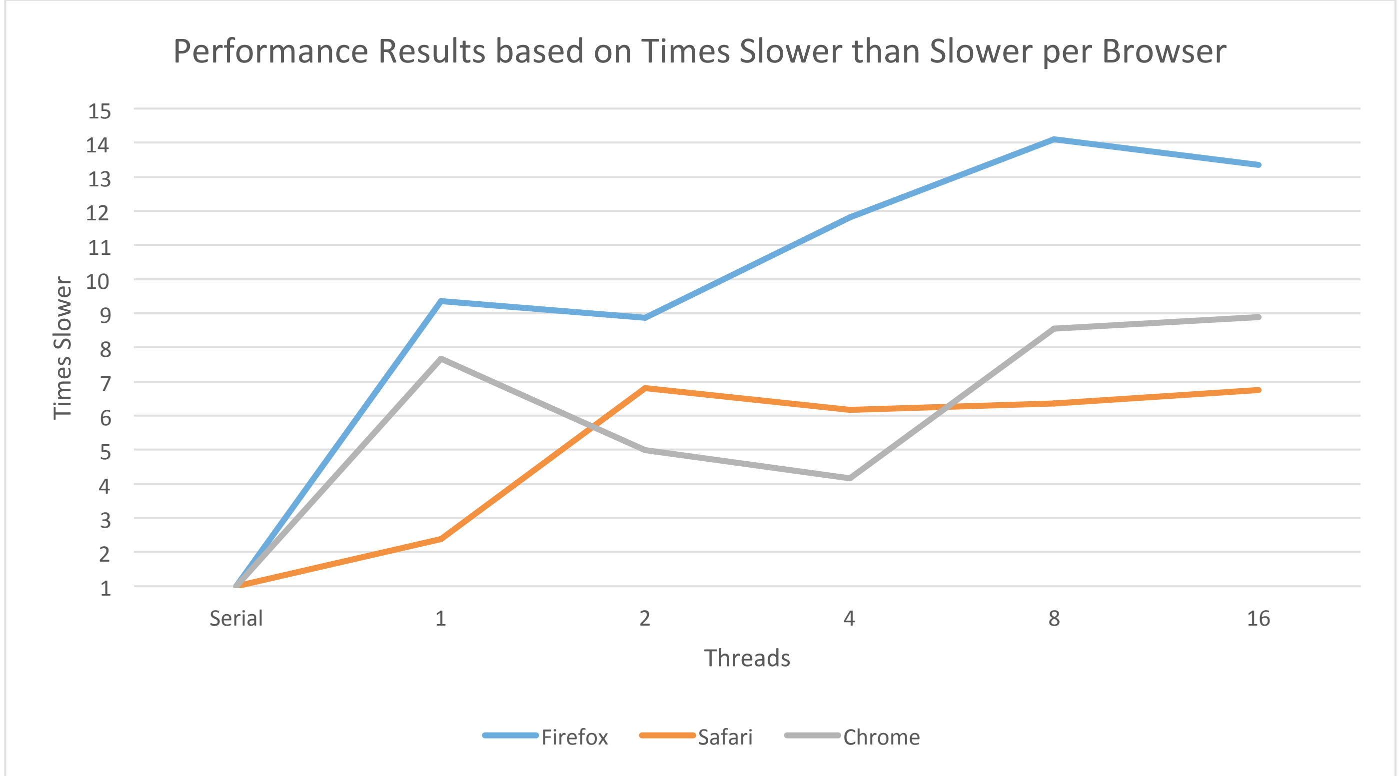
Figure 3: Code being converted, a simple for loop.

within the loop block. It renames all variables to an object within the incoming and outgoing JSON data, scrubbing the old variable names from the block. The old and new nodes are saved in a Hash Map which is utilized during the creation of the callback function to assign variables to their final values by accessing the list of variables assigned during the for-loop block. Using those assignments, the callback function uses the Hash Map of renamed variables to create the assignments.

Once this has finished, the modified for-loop is placed into a function which can be sent a single argument, the JSON data. This will be added to the main body of the code and be processed back into JavaScript along with all other processes.

C. Results of analysis

The result of breaking down one for-loop into parallel code during this method generates a for-loop that iterates over the number of cores the client has, a work function that processes the computations of the original for-loop, and a callback function which assigns the values back to the variable. The results of the simple for-loop in Figure 3 is shown in Figure 4. As can be seen, the amount of code added is significant, but the conversions are clear.



D. Problems

1) Nested for-loops

The compiler cannot fully recognize nor handle nested for-loops. This was largely outside the scope of this work and comes with some issues of its own.

2) No blocking

When a for-loop has been made parallel, it does not block subsequent, dependent code. If a variable is changed within the for-loop and used after it, the variable may contain the incorrect value.

3) Objects in arrays

Objects in arrays are difficult for the Compiler Pass to handle. Largely, they will be ignored or written to or from incorrectly.

IV. Performance

On average, the performance was much slower than serial code. The performance tests are discussed below. A significantly large computation was done which took a few square roots of large numbers in order to slow performance.

A. Performance Results

The performance tests are shown above. The parallel code was not faster than serial code even one time. CPU utilization did show that more than one core was being utilized, but often not enough to meet the threads specified. For example, eight threads would be spawned, but only 4 would be used.

B. Data Analysis

The performance indicates browser-dependent issues with Web Workers. Safari was able to maintain six times slower performance with the number of threads, but Firefox had jumps and dips in its performance which indicate arbitrary thread scheduling. Some of this slowdown is due to the reassigning of data, but this should not accumulate slowdowns in the way the results indicate.

V. Conclusion

JavaScript can be statically analyzed and made parallel by utilizing the correct tools. However, the performance tests suggest that Web Workers and the Internet may not be ready for fully parallel JavaScript. More advancements are needed in this area before work like this can be fully utilized or built upon to become useful for developers to begin using.

VI. Bibliography

[1] N. Bliss, "Addressing the Multicore Trend with Automatic Parallelization," *Lincoln Laboratory Journal,* vol. 17, no. 1, pp. 187-198, 2007.

[2] M. Mehrara, P.-C. Hsu, M. Samadi and S. Mahlke, "Dynamic Parallelization of JavaScript Applications

Using an Ultra-lightweight Speculation Mechanism," *IEEE,* pp. 87-98, 2011.

[3] E. Fortuna, O. Anderson, L. Ceze and S. Eggers, "A Limit Study of JavaScript Parallelism," 2010. [Online]. Available: https://homes.cs.washington.edu/~luisceze/publications/fortuna-iiswc2010.pdf. [Accessed 3 November 2015].

[4] "Web Workers," W3C, 24 Sep 2015. [Online]. Available: https://www.w3.org/TR/workers/#references. [Accessed 20 May 2016].

[5] K. Dewey, V. Kashyap and B. Hardekopf, "A Parallel Abstract Interpreter for JavaScript," in *IEEE/ACM International Symposium on Code Generation and Optimization*, Santa Barbara, 2015.

[6] H. Zhong, M. Mehrara, S. Lieberman and S. Mahlke, "Uncovering Hidden Loop Level Parallelism in Sequential Applications," University of Michigan, Ann Arbor, MI.

[7] A. Savitzky and S. Mayr, "Parallel.js: Parallel Computing with JavaScript," [Online]. Available: https://adambom.github.io/parallel.js/. [Accessed 20 May 2016].

[8] D. Vardoulakis, "High level overview of a compilation job," Google, 8 Mar 2016. [Online]. Available: https://github.com/google/closure-compiler/wiki/High-level-overview-of-a-compilation-job. [Accessed 11 May 2016].

[9] N. Santos, "Closure Compiler AST Documentation," 22 Sep 2010. [Online]. Available: https://closure-compiler.googlecode.com/files/closure-compiler-ast.pdf. [Accessed 19 May 2016].

[10] M. Zhou, "Links to Closure Compiler design documents and proposals," Google, 15 Feb 2016. [Online]. Available: https://github.com/google/closure-compiler/wiki/Design-Documents. [Accessed 11 May 2016].