JSProf - Javascript Profiler written in Javascript

Sneha Shankar Narayan, Vijay Pasikanti

October 21, 2013

CS630 - Graduate Systems

Contents

Chapter 1

Design and Approach

After looking at the existing profilers out there namely Google Chrome's developer tools and the Firebug suite we decided to give two options for the developers out there, namely a static way to analyze their Javascript files and also a dynamic way to do the same. We have used Esprima to parse the javascript code. In order to take care of the execution times and analyzing the call tree we have instrumented functions to profile the start and end of the function definitions. In order to take care of asynchronous callbacks we have replaced the function definitions of callbacks as well as the calls to the callbacks to have extra arguments. Using these arguments we track the time taken by the asynchronous functions and the callbacks. When the code is run we update few global data structures that captures the required data like the statistics of the function calls made and also the call tree that is generated.

1.1 Functionality provided

The following functionality has been provided.

1.1.1 Execution times for individual functions

Using the instrumented start and end functions we calculate the time taken by each function that is invoked.

1.1.2 Edges between caller and callee functions

By parsing the input code that we get from the user we make a list of all the functions that are called and the names of their callees. We also build a tree that shows what all functions are invoked and in what order they are invoked.

1.1.3 Frequency of calls

We get this information from the global function tree that we update when the code runs.

1.1.4 Reconstruction of the dynamic call paths

We present this information in the form of a function tree that shows the order in which that the functions are invoked.

1.1.5 Identification of hot paths

We define hot paths as the paths that take the longest time to execute. For each piece of code that is profiled by our profiler we find out the path that takes the longest execution time and we output that path to the user such that the user can optimize that path.

1.1.6 Tracking sources/causes of asynchronous callbacks.

We change the definition and the calls made in asynchronous callbacks to add an extra argument, which is the start time of execution of the caller, this information is used by another instrumented function that is present in the callback to determine the actual execution time of the asynchronous function.

1.2 Static analyzer

The static analyzer deals with the javascript files that are input to the profiler. Here the user can input a javascript file to the user interface provided on the browser. The javascript files are parsed and the resulting structure is shown to the user on the browser interface.

1.3 Dynamic analyzer

The dynamic analyzer provides a way to analyze any website. In the user interface provided there is a textbox present that can be fed the URL of any website that the user wants to profile. A proxy server will be hosted on the node that the profiler runs in and this proxy server intercepts the code from the actual server and also instruments the code with the required instrumentation and the browser reads this instrumented code. The user is

also provided a button to start and stop the profiling. While the profile is being run the user has the freedom to use the website the way he normally would. After the user decides to stop the profiling he can see the results with all the functionality provided in the user interface.

Chapter 2

Results

The results are provided in the same user interface that the user uses to input the file/website that has to be profiled.

2.1 Interpreting the results

- **Frequency of calls:** The hits specified is the number of times that function is invoked.
- **Execution times of individual functions:** This captures the average execution time that each function takes.
- **Edges between caller and callee functions:** This shows all the caller-callee pairs that have been invoked in the function and also the number of times that each caller-callee pair has been invoked.
- **Maximum execution time:** This shows the maximum execution time that is taken by one of the paths in the function call tree.
- **Hot path:** This shows the path that has taken the maximum time to execute in the function tree.
- **Function table:** This table shows the execution time taken by each function branch and also the time taken by the function in executing it's own tasks (not including the execution time taken by the functions that the current function invokes) The table also shows the level of each function.

Chapter 3

Testing

This chapter briefly describes the testing that was done on the profiler.

3.1 Test cases

We performed various tests on the static and the dynamic analyzer to take care of all the conditions that we could think of.

3.1.1 Static analyzer

All the test cases that were used are under the /tests directory. Test cases considered were of the form

- Code that had a lot of function definitions, with functions being defined inside other functions.
- Code with funny indentation.
- Code with return statements strewn around in various places. Here we had to take care of the end of function instrumentation.
- Code with asynchronous callbacks. Instrumentation had to be done carefully here.
- Code with recursive and iterative calls.
- Code object definitions.

3.1.2 Dynamic analyzer

Here we gave various URLs to our analyzer. Following are the issues which we have tested

- Code with lot of functions and javascript files.
- Code with javascript embedded in the HTML.
- Code with lot of asynchronous callbacks.
- Code with HTML comments embedded in the ¡¡script¿¿ tag.
- Issues with the instrumentations not working properly.

Following are some of the URLs that we have tested our code on.

- www.umass.edu
- www.cs.umass.edu

Chapter 4

Setup

This chapter describes the setup required to run the profiler

4.1 Requirements

- Node.js
- Any browser (Preferably Google Chrome or Mozilla Firefox)

4.2 npm packages that are required

- esprima
- escodegen
- estraverse
- path
- http
- url
- request
- mime
- http-proxy
- express

4.3 Steps to setup proxy server

- Go to the nodejs command prompt and change the directory to the one where you have stored the JSProf source code. Run the file server.js.
- Change the browser settings to always fetch data from the proxy server configured at "localhost:9000".

4.4 Using the profiler

- Go to the browser and type localhost:9000 on the address bar. The user interface provided appears and you are good to go!