UNIVERSITY OF MASSACHUSETTS AMHERST- COMPUTER SCIENCE-FALL 2013

JSProf: A Javascript Profiler

Authors: Eswer Kishore Kumar, Nimish Gupta

October 21,2013

1 INTRODUCTION

The aim of this project is to build a Javascript profiler written in Javascript. It is supposed to be browser independent. The profiler is expected to report the following information

1. Execution times for individual functions
2. Edges between caller and callee functions
3. Frequency of calls
4. Reconstruction of the dynamic call paths
5. Identification of hot paths
6. Tracking sources/causes of asynchronous callbacks

Original problem statement can be found at: http://plasma.cs.umass.edu/emery/grad-systems-project-1

2 APPROACH

The basic flow of our application is like this. Individual stages are discussed in detail later.
Instrument code –> Collect Run Time data –> Post Process –> Show Results

3 Instrument Code

To collect the required information we need to have our special instructions in specific places in code. To collect and show the above information we need to track each function entry and function exit. For tracking asynchronous callbacks we need to look for call expressions in Javascript code. To capture function declarations and expressions in Javascript we first represent it into an annotated canonical form using a Javascript parser. We then traverse the canonical form to look for function declarations and function expressions and add our code to all functions found upon traversal. We use Esprima (http://esprima.org/) parser and Escodegen (https://github.com/Constellation/escodegen) to accomplish parsing and rewriting respectively.
Since a function can have multiple exit points, we wrap the function code into try finally block to ensure that we successfully capture function exit event.
Ex : Before instrumentation

```
function ()
<body>
```

After instrumentation

```
function ()
capturefunctionentry ();
try
<try block>
finally
capturefunctionexit ();
```

4 Collect Run Time Information

At run-time, the actual code interacts with our code that captures the run time call stack information. Since we are interested in capturing the time of function execution it is imperative that the overhead of our instrumentation code be minimal. We have taken various measures to ensure that this is the case. We collect call function call information on an array and tag it with start and stop time. The call stack information gives us edges between callers and callees, the time of execution of the edge and the number of times the call was made. As the program executes this data if not processed could easily occupy a lot of memory over a small interval of time. A program consists of fixed functions that are called arbitrary number of times. So reducing the callstack into edge information helps keep the memory overhead low. Since this will incur overhead if done in real-time so with do it asynchronously at a later stage. We rely on the fact that Javascript programs are mostly asynchronous to keep our memory footprint low.

5 Post Process

All the data we have collected, needs to be passed to be processed for results. Our post processor takes the collection of edges as inputs and forms a graph out of it, propagating the

times from edges to nodes. We have taken care of self-recursion. The resulting data structure has the following information
For every function in graph we have the following information.

1. Its caller functions (parents)
2. Children
3. Total time in ms
4. Self time in ms
5. Number of calls made to this function
6. Number of self recursive calls made

The data is presented in a tree-tabular format. We present two views to the user.

1. Top Down View
2. Bottom Up View

Top Down View: Top down view traces the execution flow of program, starting from the Javascript Engine context. The hot-path is highlighted with a distinct color. Initially the whole tree-table is collapsed to one single node. When expanded, it shows the functions which were called from the clicked function. The functions can be further clicked to know their children until we encounter a leaf node.
Bottom Up View: Bottom up view shows the list of functions along with their individual statistics. The functions comprising the hot-path are again highlighted. Upon clicking a function the tree expands to show the functions that calls the function that was clicked. This can be traced until we encounter the JS Engine context.
We have used a third party library called JQuery and JQuery tree-table to help us with the display of collapsible tree-table.

6 System Design

6.1 Proxy Server

A web page can have embedded JS and link to script files. We want the instrumentation to be transparent to the user. We have written a proxy program in node js that intercepts http requests. Instruments JS files before we send it to user. It also embeds link to our modules that collects and process the run-time data. This is done in the <head> tag to make sure that the modules are available before any JS code starts executing on the web page

6.2 Web Interface

We ask the user to enter the URL which he would like to profile, make the request to the web-page (through proxy server). We use an Iframe based UI to open the source page for user to interact with it. On top of the page we have a button that shows the user the result of execution. Since the pages are cross domain. We use message passing to communicate with our embedded module in the Iframe to pass the requests and profiling data for post processing.

6.3 Testing

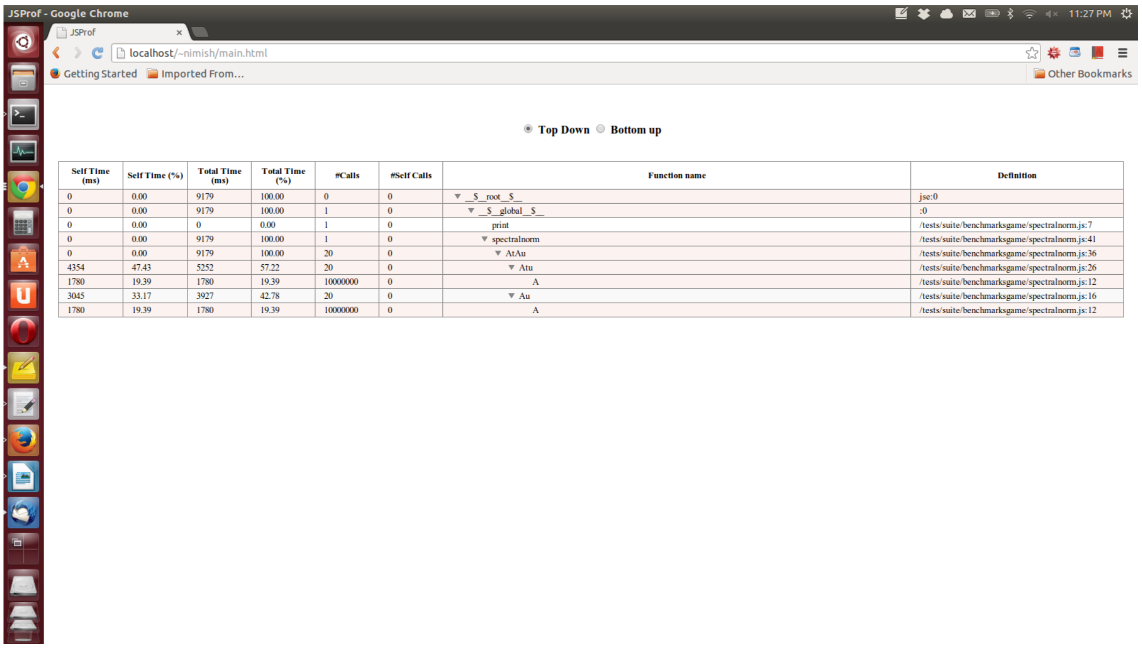
Almost all the modules have been developed in node js compatible JS, but they don't use any node specific feature. Since proxy is server side thing we have used written it entirely in node. Additionally we are using the browserify (http://browserify.org/) module of node to convert the node modules to browser side JS. We have tested the individual modules in node using the few benchmarks (http://benchmarksgame.alioth.debian.org). Browser side testing was done with Chrome and Firefox.

7 Results

We are able to collect the required information and show it except that we are currently not tracing asynchronous callbacks. There is a slight overhead incurred while running the instrumented programs. We have been using http://esprima.org/demo/functiontrace.html as our reference and we have generally observed that the overhead involved is 15-20

8 References

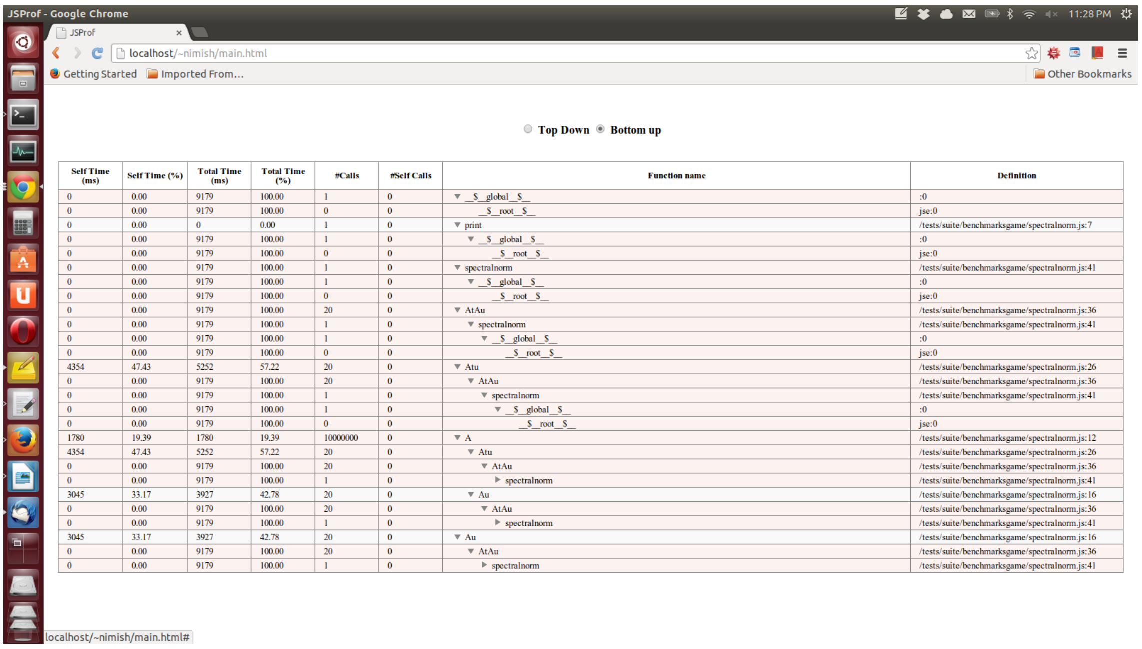
1. Esprima JS Parser (http://esprima.org/)
2. Escodegen (https://github.com/Constellation/escodegen)
3. Jquery (http://jquery.com/)
4. Jquery tree-table (http://plugins.jquery.com/treetable/)
5. Node (http://nodejs.org/)
6. Browserify (http://browserify.org/)
7. Test Benchmarks (http://benchmarksgame.alioth.debian.org)
8. Esprima function trace (http://esprima.org/demo/functiontrace.html)



● Top Down ○ Bottom up

Self Time (ms)	Self Time (%)	Total Time (ms)	Total Time (%)	#Calls	#Self Calls	Function name	Definition
0	0.00	9179	100.00	0	0	▼ __$__root__$__	jse:0
0	0.00	9179	100.00	1	0	▼ __$__global__$__	:0
0	0.00	0	0.00	1	0	print	/tests/suite/benchmarksgame/spectralnorm.js:7
0	0.00	9179	100.00	1	0	▼ spectralnorm	/tests/suite/benchmarksgame/spectralnorm.js:41
0	0.00	9179	100.00	20	0	▼ AtAu	/tests/suite/benchmarksgame/spectralnorm.js:36
4354	47.43	5252	57.22	20	0	▼ Atu	/tests/suite/benchmarksgame/spectralnorm.js:26
1780	19.39	1780	19.39	10000000	0	A	/tests/suite/benchmarksgame/spectralnorm.js:12
3045	33.17	3927	42.78	20	0	▼ Au	/tests/suite/benchmarksgame/spectralnorm.js:16
1780	19.39	1780	19.39	10000000	0	A	/tests/suite/benchmarksgame/spectralnorm.js:12

Figure 7.1: Top Down View



○ Top Down ● Bottom up

Self Time (ms)	Self Time (%)	Total Time (ms)	Total Time (%)	#Calls	#Self Calls	Function name	Definition
0	0.00	9179	100.00	1	0	▼ __$__global__$__	:0
0	0.00	9179	100.00	0	0	__$__root__$__	jse:0
0	0.00	0	0.00	1	0	▼ print	/tests/suite/benchmarksgame/spectralnorm.js:7
0	0.00	9179	100.00	1	0	▼ __$__global__$__	:0
0	0.00	9179	100.00	0	0	__$__root__$__	jse:0
0	0.00	9179	100.00	1	0	▼ spectralnorm	/tests/suite/benchmarksgame/spectralnorm.js:41
0	0.00	9179	100.00	1	0	▼ __$__global__$__	:0
0	0.00	9179	100.00	0	0	__$__root__$__	jse:0
0	0.00	9179	100.00	20	0	▼ AtAu	/tests/suite/benchmarksgame/spectralnorm.js:36
0	0.00	9179	100.00	1	0	▼ spectralnorm	/tests/suite/benchmarksgame/spectralnorm.js:41
0	0.00	9179	100.00	1	0	▼ __$__global__$__	:0
0	0.00	9179	100.00	0	0	__$__root__$__	jse:0
4354	47.43	5252	57.22	20	0	▼ Atu	/tests/suite/benchmarksgame/spectralnorm.js:26
0	0.00	9179	100.00	20	0	▼ AtAu	/tests/suite/benchmarksgame/spectralnorm.js:36
0	0.00	9179	100.00	1	0	▼ spectralnorm	/tests/suite/benchmarksgame/spectralnorm.js:41
0	0.00	9179	100.00	1	0	▼ __$__global__$__	:0
0	0.00	9179	100.00	0	0	__$__root__$__	jse:0
1780	19.39	1780	19.39	10000000	0	▼ A	/tests/suite/benchmarksgame/spectralnorm.js:12
4354	47.43	5252	57.22	20	0	▼ Atu	/tests/suite/benchmarksgame/spectralnorm.js:26
0	0.00	9179	100.00	20	0	▼ AtAu	/tests/suite/benchmarksgame/spectralnorm.js:36
0	0.00	9179	100.00	1	0	▶ spectralnorm	/tests/suite/benchmarksgame/spectralnorm.js:41
3045	33.17	3927	42.78	20	0	▼ Au	/tests/suite/benchmarksgame/spectralnorm.js:16
0	0.00	9179	100.00	20	0	▼ AtAu	/tests/suite/benchmarksgame/spectralnorm.js:36
0	0.00	9179	100.00	1	0	▶ spectralnorm	/tests/suite/benchmarksgame/spectralnorm.js:41
3045	33.17	3927	42.78	20	0	▼ Au	/tests/suite/benchmarksgame/spectralnorm.js:16
0	0.00	9179	100.00	20	0	▼ AtAu	/tests/suite/benchmarksgame/spectralnorm.js:36
0	0.00	9179	100.00	1	0	▶ spectralnorm	/tests/suite/benchmarksgame/spectralnorm.js:41

Figure 7.2: Bottom Up View