

Javascript Language Extensions

Esha Choukse, Lara Schmidt

I. INTRODUCTION AND PROJECT GOALS

V8 is a compiler for javascript that is designed to make javascript as fast as possible. It has many optimizations and features. However one thing that V8 does not provide is a way for a developer or user to provide 'hints' to the compiler. Optimizing takes time and V8 doesn't want to lose time optimizing when it doesn't need to. For example it won't optimize a function until it is 'hot'. So therefore, by allowing the compiler to take hints about whether or not it should optimize a function, we can allow V8 to gain even more speed.

II. API

We implemented three optimizations to prototype our ideas. A developer could specify the optimization to use by adding code into the specific javascript. The developer defines an optimization by function. This was because V8's optimization and code-tracking is done per function. The first optimization was Optimize Immediately. This optimization would optimize a function immediately upon it's creation. The second optimization was Never Optimize. This would cause a function to never attempt to be optimized. The third optimization is 'Edge Weight Profiling' and is more complicated and has two parts (and two corresponding user inputs). This optimization ended up being fairly complicated so we will explain it later.

The programmer would specify these optimizations inside a multiline comment with a special header. They would also specify the function name and the respective optimization. An int was used to make parsing easier. 0 for Optimize Immediately, 1 for Never optimize and 2 and 3 for Edge Weight Profiling. The two for Edge Weight Profiling have user inputs associated with them. For example:

```
/*LEZ
fname1,0;
fname2,2,100;
fname2,3,5;
fname3,1;
*/
```

In this example fname will be optimized immediately, fname2 will have the Edge Weight Profiling optimization with input of 100 and 5, and fname3 will be never optimized. One could give several optimizations to one function by just putting the function name in several lines as with fname2.

We also wanted to provide an ability for the user to optimize the javascript without the developer. So we

created a Chrome browser extension that would insert the multi-line code into a viewed html file and cause our optimizations to be run. The browser inserted the javascript comments into a script tag at the beginning of html pages before the page was ran, allowing it to be reached first by V8. For a prototype we just allowed the user to have a set of saved optimizations that was then inserted into pages on a certain domain, however one can see how it could be extended to allow the user to specify different optimizations for different pages.

III. EDGE WEIGHT PROFILING

The edge weight profiling optimization is complicated, so we will explain it here. While ideally we would like to have a better user-facing optimization, we ran into trouble with the complexity of V8. However this does give the developer more power. By default Chrome will self optimize it's own function after certain conditions are hit. For example it runs X times or there is a loop with Y runs in it. The way it does this is to set a counter to a set value. By default it is set to 6144 and this as well as all other constants we mention have a modifiable whole-system V8 flag. The counter decrements whenever a function returns (to count the times a function is run) and when a backedge of a loop is taken.

The weight of the default return decrement is set to 6144/130. This will for example cause a function to optimize after 130 iterations. Our first part of this optimization allows the user to set the 130 to something else. Therefore if a user sets the number lower, it will optimize quicker.

The loop counter by default decrements by 1 or by amount of code in loop / 170, whichever is larger. Our second optimization allows the user to modify the weight of this counter. Therefore a larger weight will optimize faster.

However though this logic is very evident in the code, we are getting some weird behavior from Chrome that we have not been able to figure out with limited time. For example sometimes Chrome will optimize a function that runs 1000 times. But if it has a while loop (that even happens very few times) it will not optimize even after 1000 runs. Also sometimes we saw inconsistencies even in this. This doesn't make any sense. It seems that maybe Chrome has a bug or some other limitation here or is trigger some other limit. However our counters do affect the optimization. For example a high loop counter will cause it to always optimize anything with any loop. And if there is no loop, our code will optimize after whatever number the user picked runs.

IV. IMPLEMENTATION

The implementation of these optimizations is actually fairly simple. However V8 is a heavily regimented and organized code-base and we will explain the difficulties with this in the next section. In V8 there is a structure called the Isolate. The isolate seems to be a representation for a thread in V8 or when used by Chrome a 'tab' in the Chrome browser. It appears to have it's own heap and garbage collection. In our code we use the Isolate as similar to 'global state'.

Our implementation first searches the first three characters of all multiline comments looking for the signature 'LEZ'. If this is found, it calls our code to begin parsing. It then adds these code into the isolate (which we added implementation to access the isolate at this point). We store a map in the isolate with the key of function name and the value a pointer to a malloced int *. The first int in the array is a bit map where each bit corresponds to whether an optimization is 'on' or 'off'. The other spots in the array are for extra information like for example the user input in the Edge Weight Profile optimization. The locations of these values are hard-coded as defines to make it easy to access.

The next part was fairly tricky as we wanted to avoid having to do string based map lookups every time we wanted to find out if a function needed to be optimized. So we solved this problem by attaching the optimization data to a per-function data structure. There were several options: Code object, JSFunction object, and SharedFunctionInfo objects. We decided to add it to SharedFunctionInfo because this object had a lot of information stored with it and was available from the JSFunction object. Also Code and JSFunction objects seem to be recreated and deleted often. We copied the int array into a structure that was used inside V8 to avoid lots of issues. This was done on creation of the SharedFunctionInfo and the information was pulled from the isolate with a map lookup. Note that we do have to do a hash map lookup once for every function even if it is not optimized.

Once we had the data inside the SharedFunctionInfo object we could use it to actually make the optimizations. For the Optimize Always optimization we made it do the optimization immediately by modifying a check in V8 that optimizes if the v8 'always_opt' flag is on. To never optimize we tricked V8 into thinking that it shouldn't optimize by setting a reason in the SharedFunctionInfo to have deoptimized. For the third optimization, Edge Weight Profile, we modified code that wrote assembly code that set the weight value of the counters to look for the values from the object we added to SharedFunctionInfo.

Another thing we implemented was to add a flag to turn our optimizations on and off. If the flag indicates not to use our optimizations, no parsing is done and a lookup into the map does not happen and no checks to the object in SharedFunctionInfo. This should allow us to test our implementation's impact and test the opti-

mizations compared to none.

V. DIFFICULTIES

We ran into many difficulties with V8. V8 is a very complex and regimented codebase with very few comments or documentation regarding more than a general overview. Here are a few of the things we had to struggle with in V8.

For one, SharedFunctionInfo, JSFunction, and Code are very regimented data structures. Their data is accessed by static offsets and their total size is used. Also there are requirements on their data that it be a heap object and it seems to create some sort of snapshot and has requirements regarding that. Also since V8 runs it's own code during compile, it would cause segfaults with no useable output.

Another difficulty with V8 is that it takes forever to compile. It took 10 minutes to compile and link a header file and on Esha's computer it took 15 minutes to even compile and link a change to a cc file.

We also struggled with the V8 garbage collector as they have special handle objects to handle garbage collected objects. This is to be able to change pointers when sweeping the objects. However these are not able to be put into a map and caused lots of issues which is why we had to use a simple malloced int[] and copy it into a V8 heap object when copying into the SharedFunctionInfo.

We ran into a lot of other issues with V8 including the Printf (that they use for logging) not always actually printing out to stdout and dealing with Chrome caching web pages on load.

Another issue we had with Chrome was that it has so many pieces to the optimization that it was hard to figure out exactly what it was doing. Usually it seemed to follow simple rules but sometimes if you did things like add more functions it would no longer follow them. This made it hard to test our optimizations and the effect they had on Chrome.

VI. ANALYSIS AND RESULTS

First we will explain our timing method used here. We have placed timers to measure the compile time of every function. We have also placed timers to count the time it takes to optimize a function. We also placed a timer to track the complete execution time. This should allow us to evaluate and prove that each of our optimizations works. We also timed our overhead by timing the time it took to parse the extra code and the time it took to do the map lookups. This was very little overhead however we will include it in our results. All numbers are based off an average and stddev of 6 runs. The benchmarks were all run on the same laptop with 8GB memory and i5 quad core.

A. Opt 1: Optimize Immediately

We were able to verify that this optimization worked by picking a simple javascript that would not normally be optimized in V8. The example we used is a simple function 10 times. V8 logged that it had optimized the function and our supplemental logging backed up our results. However we noticed upon further analysis that it was deoptimizing the code immediately when it first ran it. On further inspection the reason for this was that the function did not have enough information. This makes sense because before a function is run you know nothing about the types. However we are not sure why Chrome chose to optimize successfully before it deoptimized instead of just failing to optimize. To test our hypothesis, we attempted to optimize a function that just returned 3. V8 was able to optimize and did not deoptimize. This makes sense because it knows all the information regarding the function. On further analysis it will deoptimize if any variable is used on the right hand side of an assignment. We were also able to print out the reason for deoptimization in the previous code and it was most often "Insufficient type feedback for LHS of binary operation", though we also saw other issues in more complex code.

Because the limitations of this optimization are not very interesting, we did not do further analysis. However one can fairly easily implement Optimize immediately by using the edge weight profiling optimization.

B. Opt 2: Never Optimize

This optimization was the easiest to implement and test. V8 prints out that it wanted to optimize a function but couldn't and prints out the reason that we gave it and we can see that it never optimizes the function in the text output. However we also ran a benchmark to prove that our deoptimization works. We created a function that had a forloop that ran 100000 times inside it. Then we ran that function 2000 times. We compared a run where we told the function not to optimize and run that ignored our extensions and optimized.

	Unmodified	Modified
Execution	1525.2 ± 82.3 ms	2731.7 ± 195.9 ms
Compile	87.65 ± 8.7ms	101.5 ± 15.0 ms
Overhead	0	.428 ± .425 ms

The original run was a lot faster than our run because that function was optimized. This shows that our deoptimization function works.

Next we will try to find a case where our optimization actually causes faster execution. To test this, we create 9 functions that did simple operations including a small while loop of 100. We then ran each of these functions 135 times. This would cause the functions to optimize in the original case and not optimize in our case. This should save us on optimization time.

	Unmodified	Modified
Execution	1744.1 ± 246.9 ms	1600.7 ± 207.4 ms
Opt Compile	85.8 ± 17.7ms	58.72 ± 16.8 ms
Full Compile	83.55 ± 12.8	93.03 ± 14.14ms
Overhead	0	.837 ± .810 ms
Sum	1913.4 ± 231.7 ms	1753.317 ± 231.69 ms

These results are a little confusing. We can see that we gain some time on the fact that we don't optimize. However we seem to be faster for execution as well. One pattern that we noticed was that execution seemed to take a little longer when an optimization was done. We are not sure why this is. It is possible it is the cost of catching that it needs to be optimized and setting it to be optimized. However one may notice that it is only around 1 standard deviation higher, so it is not that much. Even with just the improvement in compile time, this optimization would be successful.

C. Opt 3: Optimized after k runs

We tested that this optimization works by running many different kinds of codes and verifying that they optimized when normally they wouldn't.

A good use of this optimization is if your function runs just enough times to be optimized. In this case default V8 would optimize but then quickly end and not get a good result. Our test case was 20 of the same function. Each function did a few simple calculations and a while loop that ran to 30. Each function was run 550 times. We gave the first input as 10 and the second input as 5. This should cause our function to optimize very quickly. The results are shown below.

	Unmodified	Modified
Execution	1217.4 ± 70.5 ms	1126.9 ± 44.8 ms
Compile	99.4 ± 9.3 ms	101.8 ± 8.1 ms
Overhead	0	0.583 ± 0.54 ms
Sum	1316.8 ± 78.2 ms	1229.24 ± 48.8 ms

As you can see, both default and ours optimize. However we have less execution time because our optimization was done earlier.

VII. WHAT WE LEARNED

One of the big things that was interesting to see was how V8 worked compared to the compilers we have talked about in class. It was interesting to see things we had learned in class in the V8 engine, and also how it was different. For example how their garbage collector uses Handles as was mentioned in the slides. It was also interesting to see how they actually implemented edge profiling and how they handled deoptimization and optimization and the different heuristics that were used. It is likely that these heuristics went through a lot of testing to arrive where they are now.

VIII. LIMITATIONS

There are a few limitations with our implementation. One is that we currently just place our function names into the isolate. This means that these function names will be loaded on future pages that are loaded in that tab. This is a performance issue (if too many people use it it will make the stored data larger) and also a security issue as it allows a page to slightly influence other pages that are loaded.

Another issue is that Chrome’s optimizations are very complex and we were unable to test infinite test cases. We know our optimizations work for simple cases but they might not be as effective for larger files.

Another issue is that all our tests were run on initial page loads. However V8 can cache code between runs so

our optimizations may not help as much if a user were to refresh a page.

IX. CONCLUSION

All in all our optimizations show that a little gain can be made from allowing user and developers to add their own hints to the compiler. We were unable to generate a lot of gain but it is possible with more knowledge of V8 that this would be a reasonable course of action. However V8 does do a really good job of guessing the optimizations itself so it may not be worth the hassle of implementing the code (safely) and putting out API and getting people educated on this new ability.

[1] Example if we need it