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. There have been a few papers in the past that have explored similar things, [], however not ones that have explored in particular V8 and javascript optimizations.

II. API

We implemented three optimizations to prototype our ideas. A developer could implement these by adding code into the specific javascript file. The deverloper 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 immediatly upon it's creation. The second optimization was 'Never Optimize'. This would cause a function to never attempt to be optimized. The third optimization was 'Optimize After X runs'. This would cause the function to be marked for optimization after X runs.

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 for 'Optimize after X runs'. The 'Optimize after X runs' was followed by another int that specified the given X. For example:

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

In this example fname will be optimized immediately, fname2 will be optimized after 100 runs, and fname3 will be never optimized. One could give several optimizations to one function by just putting the function name in several lines, however in the case of these three optimizations it doesn't make sense as they are opposing optimizations.

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 code into the javascript 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. 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 X in 'Deopt after X'. 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 funciton 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 availabe 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 'Optimized after X' we modifyed code that wrote assembly code that set the something similar to a weighted counter to use our weight instead of a default flag.

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. This should allow us to test our implementation's overhead and test the optimizations comparead to none.

IV. 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. Also it is very hard to narrow down page changes with V8 as this is handled in chrome and it just passes pieces of the javascript to V8 when needed. This makes sense as Chrome will load pieces of a page at a time but makes it hard for us to track timings.

Another issue we had with V8 was caching. It caches code between refreshes which can make it hard to test things. Sometimes it also seemed to cache data between runs as it would act differently the first time you run a file verses the second and third time. It was very strange behavior. The biggest problem we had here was analysis as it was hard to get the compiler to act consistenly.

V. 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.

To make our analysis clearer we will point out that V8 will automatically attempt to optimize a function after a certain number of consistent runs. This default flag is set to 130 though at times it seems to take more runs. Note that this 130 is what our 'optimize after X' flag changes. Default V8 will never optimize a function that runs less than 130 iterations. We use this to prove that some of our optimizations work.

A. Opt 1: Optimize Immediately

We were able to verify that this optimization worked by picking a simple javascript that ran a simple function 10 times. V8 logged that it had optimized the function and our supplemental logging backed up our 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 it optimized successfully before it deoptimized instead of just failing to optimize. We were also able to print out the reason for deoptimization in the previous code and they were things regarding not enough information or overflow. 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 had 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 futher analysis. However one can easily implement 'Optimize immediately' as something like 'Optimize after 10 runs'. TODO(laraschmidt) change this to 1 if it works but not sure if it does because no enough time.

B. Opt 2: Never Optimize

This optimization was the easiset 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.

	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 shorter 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

C. Opt 3: Optimized after k runs

D. Real Website Benchmarks

VI. WHAT WE LEARNED

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. TODO(MORE HERE)

VII. LIMITATIONS

VIII. FUTURE WORK

There is a lot of future optimizations that we could explore with the framework. There are a lot of flags that are set as constants and it would be interesting to go through them and see if any can be turned into developer-defined constants. This depends of course on whether they are used at isolate-level or function-level.

We

IX. CONCLUSION

[1] Example if we need it