Project Report: V8 Adaptive Inlining

Ishan Bhargava (ibhargav), Ethan Chu (ethanchu)

15-745

This semester, we have implemented a new inlining method in the V8 JavaScript JIT engine, based on the paper 'An Optimization-Driven Incremental Inline Substitution Algorithm for Just-in-Time Compilation'

The web page for our project can be found here: (project webpage), and contains links to the paper we implemented, as well as the source code for the project and benchmarks.

Contents

1 Introduction 2

2 Implementation Details 4

3 Experimental Setup 8

4 Experimental Evaluation 9

5 Conclusions 12

1 Introduction

Unlike ‘traditional’ optimizing compilers such as LLVM and GCC, V8 has far more constrained resources. In language such as C++, Rust, or Swift, even the simplest ‘Hello world’ program can take seconds to compile, and for more complicated projects, compile times of hours are not abnormal. These would all be completely unacceptable to V8, and therefore V8 must optimize programs while avoiding taking up too much time on its own.

V8 runs a wide range of conventional compiler optimizations on programs, such as constant propagation/folding, value numbering, strength reduction, and inlining. Contrary to popular belief, many developers write JavaScript in a fairly functional style.

```
const salesJson = DownloadData();
const totalRevenue =
   salesJson
   .map(json => Record.TryParseJson(json))
   .filter(maybeRecord => maybeRecord != null)
   .reduce((record, sum) => sum + record.totalPrice(), 0)
```

A naïve compiler would generate calls to each of `map, filter, reduce`, and each of those functions would contain further calls to the lambda functions. However, all functions involved here are quite small, and since the lambda arguments are constant at the call site, they could be inlined to great effect. This inlining enables developers to write code in a more declarative way, while not giving up too much performance.

1.1 Current Inlining Heuristic

Today, we expect to be able to do almost anything in a web browser, no matter how complex the task. Google Docs allows us to collaborate on presentations in real-time, and Microsoft's Visual Studio Code runs a full IDE in the browser. The ubiquitous nature of the browser means that V8 is one of the most finely-tuned pieces of software out there, and the current inlining heuristic is already very good.

The current V8 inlining heuristic considers functions one a time. For each call site in a function, it examines the call frequency (number of times executed over the number of times the caller is invoked). At this stage, if a called function is very small, it is always inlined. After examining all call sites, the heuristic ranks them by size and frequency and inlines functions until the total inlined code size has exceeded a fixed threshold.

1.2 Our Approach

We follow a new approach to inlining described in the aforementioned paper. The paper implemented their algorithm in the Java 'Graal' VM, but Java and JavaScript have more in common than their four-letter prefix. Although Java is statically typed and compiled ahead-of-time, the ahead-of-time compiler usually does not do any sophisticated optimizations, and instead leaves as much information in the program as possible. All functions in Java are virtual functions, due to frequent use of inheritance. This is similar to JavaScript, as JavaScript must assume that any named function call can be rebound in the middle of the program. Java also uses type information only available at runtime to optimize and specialize functions, similar to what V8 does.

The key difference between the existing inlining and the paper's heuristic is that the paper's heuristic examines the whole call tree below a function. It then is able to decide to inline clusters of functions together, by taking into account the larger context. In addition, the heuristic does not have a fixed size limit. It becomes harder for a function to meet the criteria as the inlined code size becomes larger, but it is still possible for a small and frequently called function to be inlined.

1.3 Related Work and Contributions

Although the paper already has been implemented in the Graal JVM, we did not look at any their existing code. In addition, we doubt any existing code would have been useful as we found that the most difficult parts of our project were interfacing with V8.

Our primary contributions are implementing what is essentially an interprocedural inlining framework onto V8. It is capable of dynamically exploring the call tree and inlining across multiple functions.

2 Implementation Details

Our implementation involved modifying the inliner in V8. Specifically, we essentially rewrote `js-inlining-heuristic.cc`.

Our heuristic fits into V8's `GraphReducer` framework. It exposes a `Reduce` function which is called on each node. First, we check that the node is something which can be inlined (either a call to a function or an object constructor). Then, we make sure that the call site has all the data we need to make an inlining decision, such as a reference to a

concrete function and optionally some statistics.

After we have verified that the preliminary conditions for inlining have been satisfied, we proceed with the algorithm described in the paper. There are three phases to the algorithm: expansion, analysis, and inlining.

2.1 Expansion

The first step in the algorithm is to expand the call tree to discover more candidates for inlining. Initially, we build up a `CallTree` rooted at the current function V8 has decided to optimize. We then look for functions called by the current function. This is not as easy as it would be in languages like C or C++, because in JavaScript, functions references can be dynamically altered. To account for this, V8's `JSCall` instructions take the function to be called as an operand, which may be an arbitrary computation. For our purposes, we need to be completely sure about which function will be called in order to inline, so we only add functions to the call tree if they are called via a `HeapConstant` node, as this indicates a function which is statically known to be called.

The next challenge in building the call tree was the fact that functions are stored as V8 bytecode, but optimizations are performed on V8 'sea-of-nodes' graphs. For the root function, since V8 is already optimizing it, the graph already exists, but for other functions, this is not the case. It is not really feasible to do analysis on the bytecode, because it becomes difficult to know if JSCall instructions are to a constant function, due to the fact that use-def chains are not explicit in the graph.

The paper claims to continue the expansion phase until an `expansionDone` heuristic

returns true, but unfortunately that heuristic was never described in the paper. In practice we found that most inlining benefit comes from inlining functions at a shallow depth in the call tree, so we simply stopped the expansion phase after we exceed a certain depth.

2.2 Analysis

After we have discovered enough nodes in the call tree, we are ready to analyze it. The purpose of the analysis phase is to create clusters, which will be inlined together in the final phase. As we build the call tree, we annotate it with data such as call frequencies and local cost-benefit tuples. The local cost-benefit tuple is defined as follows:

$$(\mathrm{Cost}, \mathrm{Benefit}) = (\mathsf{size}(f), \mathsf{frequency}(f))$$

Comparisons between tuples involve comparing their cost-benefit ratios. Tuples can be added elementwise.

Our `CallTree::Analyze()` function is responsible for recursively analyzing child functions in the call tree and then computing the cost-benefit tuple for the call subtree rooted at a given node. For a given node n, we order the children by their cost-benefit ratios. For each child n_i in priority order (highest to lowest), we examine $\mathsf{tuple}(n) + \mathsf{tuple}(n_i)$. If this value has a lower ratio than the n alone, then we ignore this child. Otherwise, we mark that child as being potentially part of an inlining cluster rooted at n_i.

2.3 Inlining

Finally we are ready to select functions to inline. For each node, we first apply the `canInline` heuristic to determine if we should inline the cluster rooted at that node:

$$\mathsf{ratio}(n) \geq t_1 \cdot 2^{\frac{\mathsf{size}(n)}{16 \cdot t_2}}$$

This takes into account the cost-benefit ratio, as well as the total inlined size so far, all scaled by experimentally-chosen constants. As the inlined size $\mathsf{size}(n)$ goes up, the cost-benefit ratio must be higher to justify being inlined. This means that it becomes more difficult for a clustser to be inlined as inlining progresses, but it is never impossible.

To inline a function, we would normally simply call the appropriate function on the `JSInliner` class, but because we may be inlining functions deeper than our root function's immediate children, we need to do something more sophisticated. Usually, when V8 wants to inline one function f into another g, it builds the graph of f inside of the graph of g (i.e. sharing the storage). However, our inlining framework has already built up the graphs for the child functions, since this was necessary when exploring the call tree. In addition, we may have already mutated a child function's graph if we inlined its own children. Therefore, because we have an existing graph, we needed to make major modifications to `JSInliner`. We need to shuffle all the nodes from our child graph into the parent graph. We do so by copying each node individually, and then doing a second pass to ensure that all edges point to nodes in the parent graph and not in the child graph.

3 Experimental Setup

To measure the impact of our code, we compare the performance of our code under 3 conditions.

- Reference V8 with no inlining.
- Our V8 with adaptive inlining
- Reference V8 with no modifications

V8 contains an entry point executable `d8`. We use the built-in profiling capabilities of `d8` and examine two statistics. First, as expected, we examine the overall running time of the program. Second, we examine the portion of the running time of the program spent in JavaScript code, as opposed to JIT time in C++ or garbage collection time. This recognizes the fact that there are inefficiencies in our implementation which arose out of necessity and time. Notably, when we inline one graph into another, we currently make copies of all the nodes, which could be much slower. In addition, when we analyze a child function, we have to build the graph of that function from the bytecode. We were not sure about when bytecode gets updated for a function, so therefore it was unclear how we could potentially re-use graphs when necessary. We also examine the running time with no inlining to establish a baseline to ensure that our optimization actually makes things faster.

We ran the benchmarks on a 2021 16-inch ARM-base 10-core M1 Max MacBook Pro. For the 'reference build' benchmarks, we used the latest stable version of V8 at the time of writing, which is version 10.0.139.15, running with `--prof` flag. For the 'no-inlining' version,

we added the `--no-turbo-inlining` flag. For our development version, the source code is available in our repository in the `new-inlining` branch, and was compiled in release mode.

For our benchmarks, we use some 'real world' programs which implement some actual algorithm, rather than compute some meaningless result. We use both our own programs as well as the standard V8 benchmark suite. For more JavaScript benchmarks, we took some benchmarks from the Chromium suite and increased the input sizes. We also wanted to profile our implementation against the benchmarks the paper used, but we could not locate the source code of the benchmarks provided.

4 Experimental Evaluation

Overall, we found the algorithm to produce mixed results. In many cases, the algorithm is competitive with V8's default inliner, sometimes producing faster results, and many times being close behind.

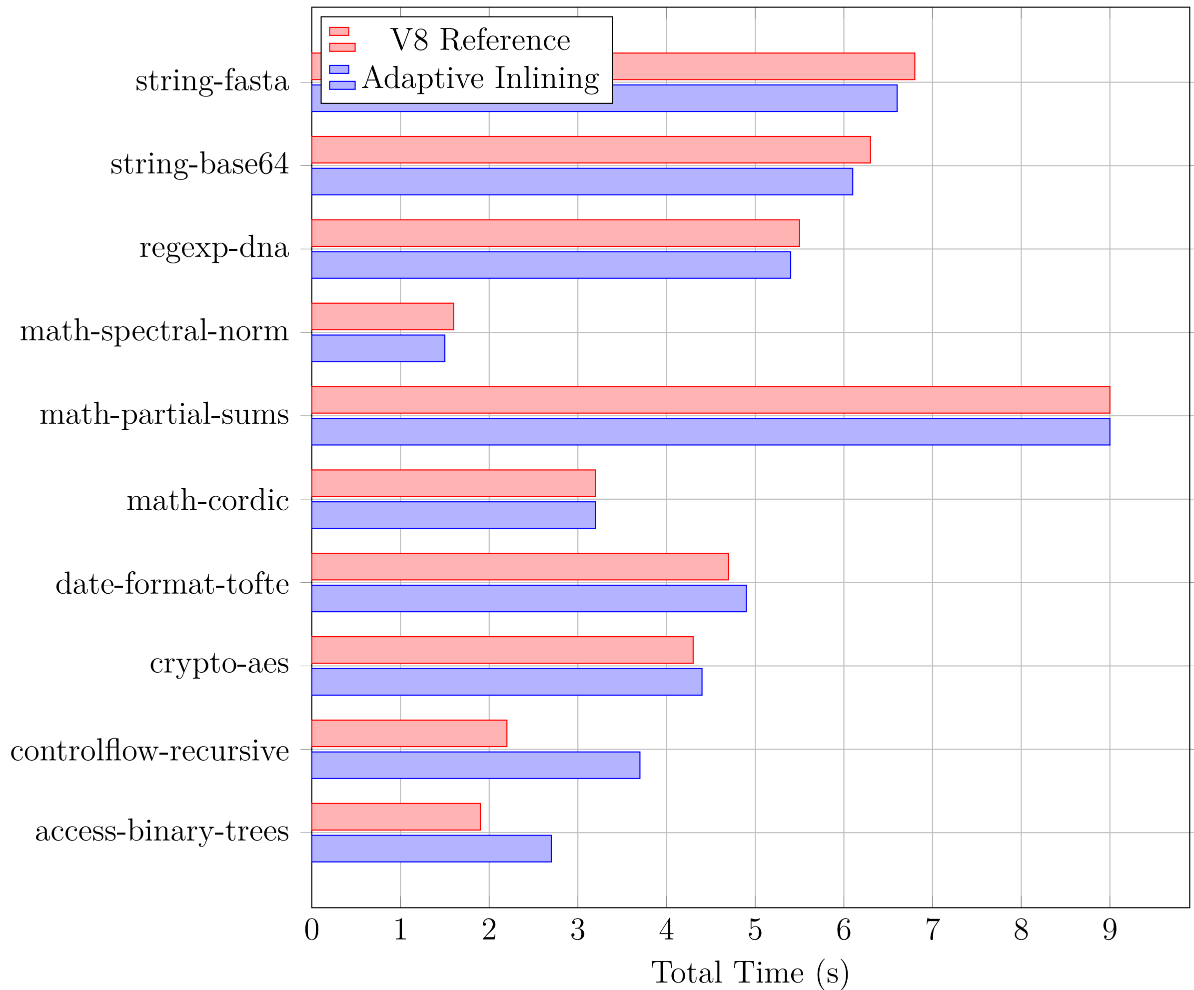

Figure 1: Sunspider Selected Benchmarks

We see that overall, our performance is close to or sometimes exceeds that of the reference V8. This makes sense, as our inlining algorithm is usually more aggressive with heavily called functions, but also takes more slightly time to run. However, the test cases that are slower are primarily the benchmarks which make heavy use of recursion. Our implementation does not inline functions recursively, which may be the cause of these outliers.

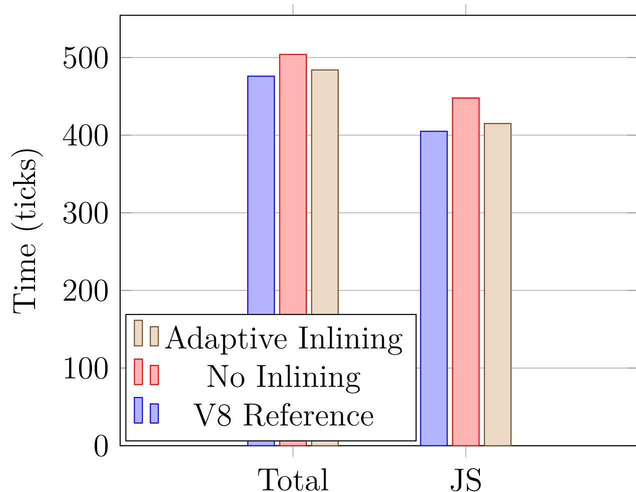

Figure 2: Linear Programming

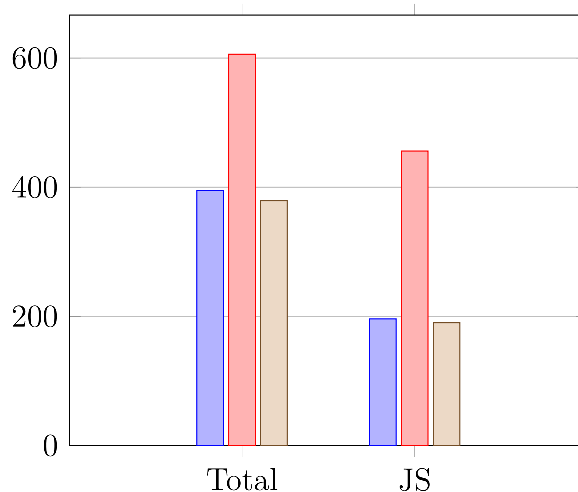

Figure 3: Fenwick Trees

Here are two more results from more sophisticated algorithms. Interestingly, we notice that test cases which have a relatively larger speedup with V8's regular inlining are even faster when run with adaptive inlining.

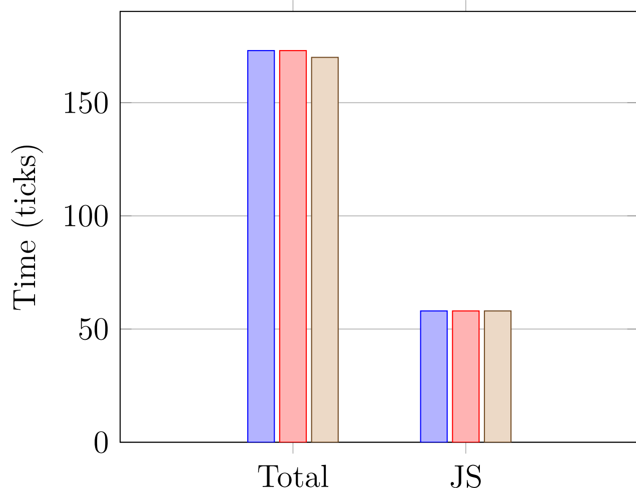

Figure 4: Gaussian Blur

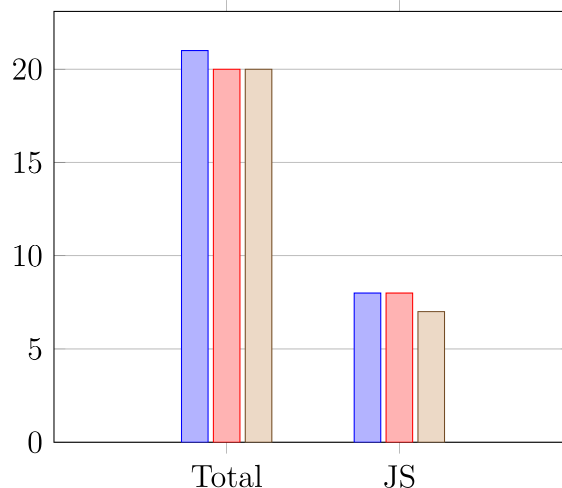

Figure 5: Encryption (`crypto.js`)

In the Chromium bechmarks, the results are quite close, with our inlining method barely taking the lead in some cases. This is consistent with our expectation that our inlining method would be close in performance to the original inlining method in most cases.

5 Conclusions

Our goal was to evaluate the approach described in the paper, and we believe that we have gathered enough information to do so.

We find that the inlining method presented in the paper is effective in most cases, but we are not sure if it is effective enough to justify its complexity. The algorithm performs well when dealing with object-oriented JavaScript code, such as those which make frequent of one-line, simple functions such as object property 'getters' and 'setters'.

The current optimizer considers functions in isolation, which is much simpler than attempting to traverse the call tree of a function and recursively inline multiple times. Currently we have grafted this interprocedural analysis support onto V8's optimizer, but implementing first-class support for this may significantly increase complexity, as well as potentially interfere with V8's ability to optimize multiple functions in parallel.

5.1 Surprises and Lessons Learned

We were rather surprised at how the paper did not describe several of the heuristics it used. This left us having to guess at what the authors intended, which makes it difficult to reproduce the code. In addition, we wish the authors would have made it easier to find the source code for the benchmarks they were using.

5.2 Future Work

We believe that our optimization itself could be further optimized to great effect. Further work would need to be done in enabling V8 to conduct analysis across multiple functions in

an efficient way. For example, we believe it would be useful to store the sea-of-nodes graphs of functions, in addition to their raw bytecode. This would be a major enhancement over our current approach where we need to rebuild the graph from the bytecode every time we need to inspect a particular function.

5.3 Distribution of Credit

We believe the credit should be evenly divided given. In order to facilitate a good level of collaboration, we met in person to do much of the work. We found this very useful since the V8 source code is quite complicated and is not extensively documented, so it was helpful to work on it together.