JavaScript Profiling and Optimization on V8

Yilin Zhang
zylime@gmail.com

C. Vic Hu
vic@cvhu.org

***Abstract*—In this course project, we want to focus on the trace profiling and optimizations in the V8 JavaScript Engine used in Google Chrome. By learning from their existing compiling infrastructure and optimization processes, we extract the key essence out of the works done by the V8 open source community, and twitch V8 with effective optimization methods. Ultimately, we want to study what it takes to build a super fast JavaScript engine in the industry, and to see if we can come up with some feasible ideas to make some enhancements.**

After collecting measurements on a selective benchmarks under various optimization conditions, we found that the optimizing compiler in V8 has relatively low impact compared to inline caching. With the addition of inline caching, the performance can be improved to 4.7 times as fast as the one without inline caching, while the improvement brought by the optimization is almost margin and sometimes even negative. These observations reinforce the important of reusing shared code structure and hot code identification to avoid unnecessary optimization overhead.

I. Introduction

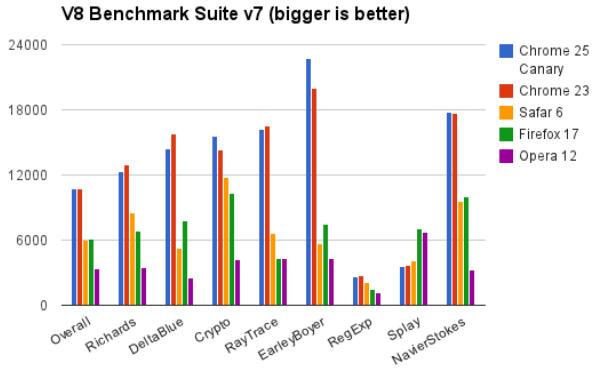
Although JavaScript is traditionally translated into bytecode by an interpreter, more and more JavaScript Engines in modern browsers are designed to compile directly into machine code. Our project will mainly focus on trace profiling [8] in V8, and making constructive adjustments according to the optimization techniques we have learned in class. We will use the SunSpider JavaScript benchmark and the V8 benchmark to measure and compare the existing infrastructures, and make a sound analysis of the results. Our overall goal is to understand the common optimization procedures performed by modern JavaScript Engines, as well as the possible performance enhancements with the knowledge we've acquired from EE382V.

In this paper, we will cover our motivations for doing this project, background information and detailed compilation processes about the V8 engine, profiling results, and comparisons to show the effectiveness of the optimizations done in V8.

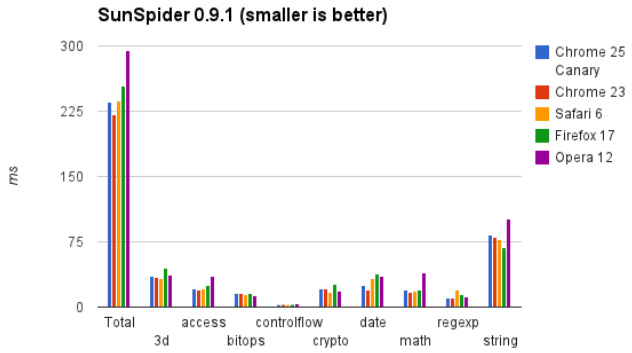
II. Motivation

JavaScript has been widely used in web-based applications to increase richer interactions and visualizations [2] since it was first supported by Netscape 2 beta back in 1995 [1]. In over fifteen years, it has evolved into a variety of frameworks and libraries to enable a more interactive and dynamic web browsing experience [3], or even to build high-performance network programs [10]. Besides its applications in web-based softwares, JavaScript has also gained its popularity from applications such as Adobe Flash, Dashboard widgets in Mac OSX, browser extensions, and web bookmarklets. Apart from its essential role in client-side interactions, JavaScript also became one of the mainstream server-side solutions in recent years [10].

As the popularity of web-based applications and services increases, browser performance has become one of the major competitions in the industry. Since JavaScript is what makes modern web pages dynamic and interactive, how to optimize its compilation/interpretation is the key component to building fast and robust modern browsers. We compared modern web browsers on two JavaScript benchmarks, including Chrome Canary, Chrome, Firefox, Safari, and Opera. In Fig.7, it was clear that both Chrome Canary (the latest beta release) and Chrome significantly outperformed any other competitors among all the test cases in the V8 benchmark suites.



(a) V8 Benchmark suite v7



(b) SunSpider Benchmark v0.9.1

Fig. 1: Testing results on 2 common JavaScript Benchmarks

In Fig.1b, although it is less obvious that Chrome was superior than its peers in the SunSpider benchmark, we can still see the dominance of V8 in general. It is due to the fact that most SunSpider benchmarks have a relatively short running time (mostly finishes in a few milliseconds), and the V8 design mainly focuses on heavily-executed applications. V8 benchmark suite was provided by the same community who developed the V8 engine, which was composed of some simulation benchmarks translated from other languages like BCPL, Smalltalk, and Scheme, as well as common operations and manipulation performance, while SunSpider mainly focused on utility performance such as text manipulation, encryption/decryption, data structure access, and common operations.

Although it has been pointed out that the testing results from these popular JavaScript benchmarks don't necessarily indicate the true performance of real-world web applications [5], the fact that Chrome dominated these competitions should somewhat reflect its success in designing a fast and efficient JavaScript engine.

III. Background

In this section, we will talk about the high-level design and implementation of the most recent V8 JavaScript Engine (v3.15.10), with specific examples to explain the key concepts and principals that make V8 outstanding.

V8 is an open source project started in late 2006 by Google, which ships with their flagship Chrome web browser. Written in C++, V8 can run both as standalone and embedded applications. Its name came from the common automobile engine, and resembled the characteristics of being fast and efficient at the same time [6].

A. Key design concepts of V8

1) Fast property access: As a dynamic programming language, object properties in JavaScript are dynamically modified in runtime, meaning that we can't just have a static memory location offset to access instance variables in programming languages like Java. In most JavaScript engines, property accesses are commonly implemented using a dynamic dictionary lookup to find the memory address, which is typically much slower and less efficient.

The concept of hidden classes is to dynamically create and change the hidden class of an object whenever a new property is added. Let's look at a straightforward location class to see how it works:

```
function Location(lng, lat){
    this.lng = lng;
    this.lat = lat;
}
```

- Initialize a hidden class C_0 for Location objects with no properties to point to
- When `lng` property is added, move the class pointer to a newly created hidden class C_1, which contains the offset of the `lng` property storage location. Update C_0 to redirect objects with property `lng` to C_1
- When `lat` property is added, move the class pointer to a newly created hidden class C_2, which contains the offset of the `lat` property storage location. Update C_1 to redirect objects with property `lat` to C_2

Using this approach, most JavaScript programs share a large portion of hidden class structures during the runtime. The advantages of using hidden classes include dictionary lookup avoidance, which speeds up the time required for property accesses, and the opportunity of leveraging optimization with inline caching, which is a classic optimization technique to effectively eliminate the overhead in dynamic typing [7].

2) Dynamic machine code: Unlike conventional interpreter for the most dynamic programming languages, V8 compiles JavaScript directly into machine code upon execution, without any intermediate byte codes. When property accessing code is

initially executed, V8 fetches the current hidden class of the corresponding object and inserts the inline caching patches along with other machine instructions.

Since V8 automatically predicts same hidden class used by all objects accessed in the same location, there might be incorrectly-patched inline caches with mismatched hidden classes, in which case V8 will handle the cache misses and redirect the object pointer to the correct hidden class. V8 also has a JavaScript regular expression engine, which was built from scratch to be automata-based and to produce machine code for regular expressions.

Both the hidden classes techniques and the machine code generation provide benefits to speed and efficiency when many objects in the code share a high likelihood of same types and usages of property accesses, which can effectively improve the JavaScript runtime performance in most programs.

3) Efficient garbage collection: Garbage collection is used to recollect memory resource that is no longer necessary and used by objects in runtime, which plays a huge role in deploying high-speed object allocation and avoiding memory fragmentation. To assemble a fast and accurate garbage collector for JavaScript compilation, V8 takes advantage of three design criteria:

- V8 employs a simple stop-the-world garbage collector, which means programs must pause execution during a garbage collection cycle. This guarantees that new objects are not allocated to a processed memory space that might become unreachable
- Because of the stop-the-world design decision, V8 should only process a portion of the object heap to reduce the impact of stopping during most garbage collection cycles
- To avoid memory leaks caused by pointer faults, V8 must knows the exact memory location of every objects and pointers

B. V8 compilation process

In V8, adaptive compilation is used to focus the optimization on the hot code that is executed again and again, and leave the rest as it is, which should improve the start-up overhead and maximum performance. To achieve this lazy optimization schema, V8 engine has the following four general components:

1) Base compiler: When the program execution first initializes, a full compiler generates a raw machine code with almost no optimization included. This process is aimed to perform fast code generation with the assumption that everything is not frequently-executed until proved otherwise during the runtime.

2) Runtime profiler: During the runtime, a system profiler is executed on the side to monitor the program execution and recognizes the portions of code that are heavily-executed and hence need to be optimized.

3) Optimizing compiler: Once the runtime profiler identifies where the hot code is, this second compiler carries out the heavy optimizations and recompiles to these targets based on the typing collected by the base compiler. During this process, static single assignment forms are used to conduct optimizations such as loop-invariant code motion, linear-scan register allocation and function inlining.

4) Deoptimization support: This feature allows the compiler to roll back from the optimized code to the original machine code generated by the base compiler, in case the hot code identification turns out to be overly optimistic. This support also gives the optimizing compiler more freedom when it is making assumptions during optimization.

Algorithm 1 *Calculate the* 25000*th Prime Number*

Require:
Ensure: The 25000th Prime Number P

```
1:  Prime list PL = {}
2:  for P = 1 to infinity do
3:      Flag = true
4:      for index = 1 to PL.size() do
5:          if P.mod(PL[i]) == 0 then
6:              Flag = false
7:              Continue
8:          end if
9:      end for
10:     if Flag == true then
11:         PL.push_back(P)
12:         if PL.size() == 25000 then return PL.back()
13:         end if
14:     end if
15: end for
```

IV. Approach

JavaScript is slow mainly due to JavaScript programs are untyped, and then compiled and run on the fly. Dynamic compilation is a great complement to static one. But completely replacing the optimized-to-death static compilation with JIT will lose the performance.

JavaScript is slower compared with other programming languages, such as C++. Before applying the optimization to the compilation of JavaScript code, we used one example to demonstrate how slow JavaScript was compared with C++.

The example here was to calculate the 25000th prime number [11]. The overall algorithm of calculating the 25000th prime number was illustrated in Algorithm1. The C++ code implementing the algorithm was in the Fig.2a while the JavaScript version was in the Fig.2b. Running these two different versions of code on the same machine showed that the runtime of C++ code was 9.6x faster than the JavaScript code. To find the reason of poor performance of the JavaScript code, we performed a profiling on the JavaScript code to determine

```
class Primes {
   public:
     int getPrimeCount() const { return prime_count; }
     int getPrime(int i) const { return primes[i]; }
     void addPrime(int i) { primes[prime_count++] = i; }

     bool isDivisibe(int i, int by) { return (i % by) == 0; }

     bool isPrimeDivisible(int candidate) {
       for (int i = 1; i < prime_count; ++i) {
         if (isDivisibe(candidate, primes[i])) return true;
       }
       return false;
     }
   private:
     volatile int prime_count;
     volatile int primes[25000];
   };int main() {
     Primes p;
     int c = 1;
     while (p.getPrimeCount() < 25000) {
       if (!p.isPrimeDivisible(c)) {
         p.addPrime(c);
       }
       c++;
     }
     printf("%d\n", p.getPrime(p.getPrimeCount()-1));
   }
```

(a)

```
function Primes() {
     this.prime_count = 0;
     this.primes = new Array(25000);
     this.getPrimeCount = function() { return this.prime_count; }
     this.getPrime = function(i) { return this.primes[i]; }
     this.addPrime = function(i) {
       this.primes[this.prime_count++] = i;
     }

     this.isPrimeDivisible = function(candidate) {
       for (var i = 1; i <= this.prime_count; ++i) {
         if ((candidate % this.primes[i]) == 0) return true;
       }
       return false;
     }
   };function main() {
     p = new Primes();
     var c = 1;
     while (p.getPrimeCount() < 25000) {
       if (!p.isPrimeDivisible(c)) {
         p.addPrime(c);
       }
       c++;
     }
     print(p.getPrime(p.getPrimeCount()-1));
  }main();
```

(b)

Fig. 2: (a) was the C++ code for calculating the 25000th prime number. (b) was the JavaScript version for calculating the 25000th prime number.

the runtime of each function. First, We executed the command in (1) to get the log file with profiling information.

```
./ia32.release/d8 samples/primes.js --prof    (1)
```

Second, (2) was applied to get the extract the runtime information of each function from the log file.

```
./tools/mac-tick-processor v8.log    (2)
```

The output of (2) provides us the runtime of each function in the JavaScript code.

Beyond our expectation, the most runtime was not spent on the main function. The main function only consumed less than 12% of the total runtime while about 30% of the total runtime was spent on the function env_access_off. With this hint, we noticed that the access of the last element, this.prime[this.prime_count], was out of the range of identified prime numbers. Though with this incorrect access, JavaScript could still give the correct 25000th prime number - 287107, the runtime increased drametically.

By correcting the access range in the isPrimeDivisible function, the new run time was only about 1.5 times of the C++ code. This improvement illustrated that JavaScript was slower tan C++, but was not much slower. The profiling with the out-of-bounds provided that more than 99% was spent on the main function.

The Fig.5 illustrated the profiling of the original JavaScript code, and Fig.6 provided the second profiling after fixing the out-of-bounds.

```
Statistical profiling result from v8.log, (10868 ticks, 81
  unaccounted, 0 excluded).
[JavaScript]:
   ticks  total  nonlib   name
   1254   11.5%   11.5%  LazyCompile: *main samples/primes.js:
  18
    959    8.8%    8.8%  LazyCompile: MOD native runtime.js:238
    643    5.9%    5.9%  Stub: CEntryStub
    468    4.3%    4.3%  KeyedLoadIC: A keyed load IC from the
  snapshot
    388    3.6%    3.6%  Stub: BinaryOpStub_MOD_Alloc_SMI
  +Oddball
      1    0.0%    0.0%  LazyCompile: ~Primes.isPrimeDivisible
  samples/primes.js:10
[C++]:
   ticks  total  nonlib   name
   3274   30.1%   30.1%  _atanhl$fenv_access_off
   1301   12.0%   12.0%  v8::internal::Runtime_NumberMod
    979    9.0%    9.0%  v8::internal::Heap::NumberFromDouble
```

Fig. 3: Profiling on Original JavaScript Code

V. Results

The experiments are conducted on an Intel Core i7 2.3GHz Mac OS X with 8GB memory. In Table I, we compare the runtime of the C++ code with the JavaScript version. The Prime here was referred to the calculation of the 25000th prime number with original JavaScript code. The Prime-2 was to calculate the 25000th prime number with optimized JavaScript code. Initially the JavaScript was 9.6x slower than the C++ version. But after code optimization, JavaScript becomed only 1.5x slower than C++. The percentage of runtime spent in the

```
[JavaScript]:
   ticks  total  nonlib   name
   1426   99.4%   99.4%  LazyCompile: *main samples/
  primes-2.js:18
      5    0.3%    0.3%  LazyCompile:
  *Primes.isPrimeDivisible samples/primes-2.js:10

[C++]:
   ticks  total  nonlib   name
      1    0.1%    0.1%
  v8::internal::StaticVisitorBase::GetVisitorId
      1    0.1%    0.1%
  v8::internal::Runtime_FunctionSetName
      1    0.1%    0.1%
  v8::internal::Map::LookupDescriptor
      1    0.1%    0.1%
  v8::internal::LAllocator::TraceAlloc
```

Fig. 4: Profiling After Optimization

main function increased from 11.5% to 99.4% in column 4 and 6 of Table I.

C++	Prime		Prime-2	
runtime (s)	runtime (s)	main	runtime (s)	main
1.3	12.5	11.5%	1.9	99.4%
	9.6x		1.5x	

TABLE I: Comparisons between C++, JavaScript and Improved JavaScript

The Fig.5 illustrated the profiling of the original JavaScript code, and Fig.6 provided the second profiling after fixing the out-of-bounds.

```
Statistical profiling result from v8.log, (10868 ticks, 81
   unaccounted, 0 excluded).
[JavaScript]:
   ticks  total  nonlib   name
   1254   11.5%   11.5%  LazyCompile: *main samples/primes.js:
  18
    959    8.8%    8.8%  LazyCompile: MOD native runtime.js:238
    643    5.9%    5.9%  Stub: CEntryStub
    468    4.3%    4.3%  KeyedLoadIC: A keyed load IC from the
  snapshot
    388    3.6%    3.6%  Stub: BinaryOpStub_MOD_Alloc_SMI
  +Oddball
      1    0.0%    0.0%  LazyCompile: ~Primes.isPrimeDivisible
  samples/primes.js:10
[C++]:
   ticks  total  nonlib   name
   3274   30.1%   30.1%  _atanhl$fenv_access_off
   1301   12.0%   12.0%  v8::internal::Runtime_NumberMod
    979    9.0%    9.0%  v8::internal::Heap::NumberFromDouble
```

Fig. 5: Profiling on Original JavaScript Code

```
[JavaScript]:
   ticks  total  nonlib   name
   1426   99.4%   99.4%  LazyCompile: *main samples/
  primes-2.js:18
      5    0.3%    0.3%  LazyCompile:
  *Primes.isPrimeDivisible samples/primes-2.js:10

[C++]:
   ticks  total  nonlib   name
      1    0.1%    0.1%
  v8::internal::StaticVisitorBase::GetVisitorId
      1    0.1%    0.1%
  v8::internal::Runtime_FunctionSetName
      1    0.1%    0.1%
  v8::internal::Map::LookupDescriptor
      1    0.1%    0.1%
  v8::internal::LAllocator::TraceAlloc
```

Fig. 6: Profiling After Optimization

Comparing the performance of V8 engine under different optimization conditions in Table II, we observed two interesting results. First, we noticed that the optimizing compiler doesn't always improve the compilation performance; in fact, sometimes its heavy overhead even deteriorates the performance. This might be caused by our selection of benchmarks, since the optimization option in V8 only shines when there are heavily executed code in a program.

The second observation from this measurement was the difference in improvements between `Prime` and `Prime-2`, where the latter is an improved version of the former. `Prime` keeps a list of all the past prime numbers, but it accesses one index over the list limit. Hence, it has a growing array data structure and requires a new type of hidden class for each iteration, which causes inline cache misses. On the other hand, `Prime-2` only accesses the last element of the list, which preserves a static array structure. Since the same hidden class for this array is being used over and over again, the power of inline caching shows a significant improvement in speed.

Benchmarks (in seconds)	Original	NoOpt	NoIC	NoOpt&NoIC
25000th Prime	22.5	22.0	95.4	103.7
25000th Prime-2	1.8	1.8	85.2	93.1
Entire V8 Benchmark Suite	21.6	21.6	81.3	79.8

TABLE II: Performance of V8 Engine on various optimization conditions

In Fig.7, runtime of the whole V8 benchmark suite was shown under different optimization setups. IC indicated that inline caching was enabled and Opt referred that optimizing compiler was applied. Comparing red and blue columns, we noticed that with or without Opt almost hit the same performance. The IC was important for V8 cause it enabled larger trunk of code optimized together. One interesting thing was that the green columns are almost always better than the orange ones. This scenario indicated that if no inline caching

was enabled, optimizing compiler became runtime overhead completely.

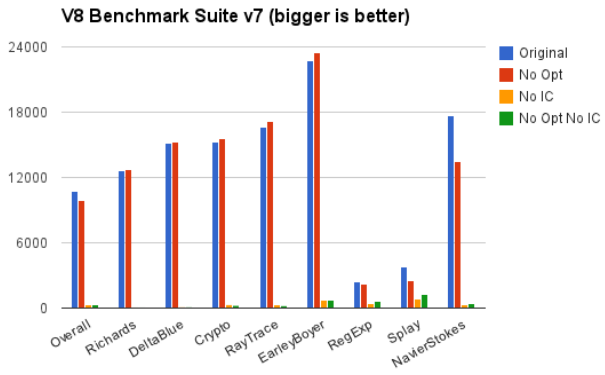


Fig. 7: V8 Benchmark suite v7

VI. Conclusion

conclusion

VII. Acknoledgment

We are very grateful to Dr. Vijay Reddi for the discussions and advices to the scope and direction of this paper.

References

[1] Eich, Brendan. "JavaScript at ten years." ACM SIGPLAN Notices. Vol. 40. No. 9. ACM, 2005.
[2] McDuffie, Tina. JavaScript Concepts & Techniques: Programming Interactive Web Sites. Franklin Beedle & Associates, 2003.
[3] Serrano, Nicols, and Juan Pablo Aroztegi. "Ajax frameworks in interactive web apps." Software, IEEE 24.5 (2007): 12-14.
[4] Tilkov, Stefan, and Steve Vinoski. "Node. js: Using JavaScript to build high-performance network programs." Internet Computing, IEEE 14.6 (2010): 80-83.
[5] Ratanaworabhan, Paruj, Benjamin Livshits, and Benjamin G. Zorn. "JSMeter: Comparing the behavior of JavaScript benchmarks with real web applications." Proceedings of the 2010 USENIX conference on Web application development. USENIX Association, 2010.
[6] Gray, James. "Google Chrome: the making of a cross-platform browser." Linux Journal 2009.185 (2009): 1.
[7] Brunthaler, Stefan. "Efficient inline caching without dynamic translation." Proceedings of the 2010 ACM Symposium on Applied Computing. ACM, 2010.
[8] Jungwoo Ha and Mohammad R. Haghighat and Shengnan Cong, A concurrent trace-based just-in-time compiler for javascript, Tech Report, 2009
[9] John Garofalakis and Panagiotis Kappos and Dimitris Mourloukos, Web site optimization using page popularity, Internet Computing, 1999
[10] "Why Everyone Is Talking About Node." Jolie O'Dell. Mashable, Inc., Web. 10 Mar. 2011.
[11] "Breaking the JavaScript Speed Limit with V8", Google I/O, 2012