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 hope to extract the key essence out of the works done the V8 open source community, and to apply the optimization techniques covered in our class. 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.**

I. INTRODUCTION

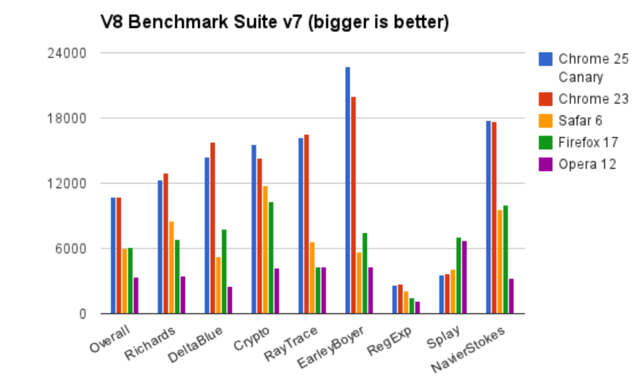
Although JavaScript is traditionally translated into byte code by an interpreter, more and more JavaScript Engines in modern browsers are designed to compile directly into machine code. Our project will mainly focus on trace profiling [7] in V8, and making constructive adjustments according to the optimization 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.

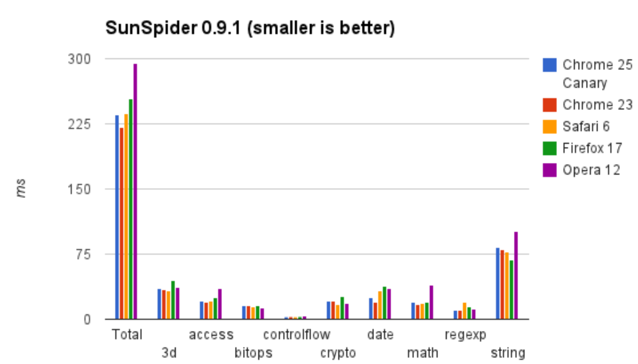
II. MOTIVATION

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 fiften 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 [9]. 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 [9].

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.1a, 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. In Fig.1b, although it became less obvious that Chrome was superior than its peers in the SunSpider benchmark, we can still see the dominance of V8 in general. 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.



(a) V8 Benchmark suite v7



(b) SunSpider Benchmark v0.9.1

Fig. 1: Testing results on 2 common JavaScript Benchmarks

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.

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 function to see how it works:

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

2) Dynamic machine code: V8 has a JavaScript regular expression engine, which was built from scratch to be automata-based and to produce machine code for regular expressions.

3) Efficient garbage collection:

B. V8 compilation process

1) Base compiler:

2) Runtime profiler:

3) Optimizing compiler:

4) Deoptimization support:

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.

Algorithm 1 *Calculate the 25000th Prime Number*

```
Require:
Ensure: The 25000th Prime Number P
Prime list PL = {}
for P = 1 to infinity do
    Flag = true
    for index = 1 to PL.size() do
        if P.mod(PL[i]) == 0 then
            Flag = false
            Continue
        end if
    end for
    if Flag == true then
        PL.push_back(P)
        if PL.size() == 25000 then return PL.back()
        end if
    end if
end for
```

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++.

```
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) C++

```
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) JavaScript

Fig. 2: the code of calculating the 25000th prime number

The example here was to calculate the 25000th prime number [10]. 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 5x 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 the runtime of each function. First, We executed the command in (1) to get the log file with profiling information.

$$./out/ia32.release/d8samples/primes.js--prof \quad (1)$$

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

$$./tools/mac-tick-processorv8.log \quad (2)$$

The output of (2) provides us the runtime of each function in the JavaScript code, as in Fig.3.

```
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

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 dramertically.

By correcting the access range in the isPrimeDivisible function, the new run time was only about 1.17 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 fixed was in Fig.4. From the new profiling, we could observe that more than 99% was spent on the main function.

```
[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

V. Results

VI. Conclusion

VII. Acknoledgment

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] Jungwoo Ha and Mohammad R. Haghighat and Shengnan Cong, A concurrent trace-based just-in-time compiler for javascript, Tech Report, 2009

[8] John Garofalakis and Panagiotis Kappos and Dimitris Mourloukos, Web site optimization using page popularity, Internet Computing, 1999

[9] "Why Everyone Is Talking About Node." Jolie O'Dell. Mashable, Inc., Web. 10 Mar. 2011.

[10] "Breaking the JavaScript Speed Limit with V8", Google I/O, 2012