Efficiency of Generated ES5 Code From BabelJS vs Closure Compiler

Ethan Bell
Ethan M Bell's CodeLump

June 25, 2015

Abstract

New features in ECMAScript6 (ES6) are very useful for developers, but are not yet implemented in enough javascript engines/runtimes to be practical to implement while still desiring compatibility. To remedy this, ECMAScript 6 precompilers were created, some of the most popular among these being Babel and Closure. These precompilers transpile ES6 code into the more compatible ES5 code. In this study, the same ES6 code was transpiled using both Babel and Closure into ES5 code, which was then tested for speed through JSPerf. The initial hypothesis that Babel's outputted code is faster was confirmed. It is therefore advisable to use Babel for ECMAScript 6 precompilation until such time as ES6 becomes more widely implemented in javascript engines.

Contents

1 Introduction

1.1 Purpose

The purpose of this experiment is to determine which ECMAScript 6 (ES6) precompiler, Babel or Closure, generates the most efficient ECMAScript 5 (ES5) code, as measured by speed of execution in the V8 javascript engine, with V8 being chosen due to its dominance of the javascript execution market (StatCounter). These precompilers allow developers to utilize the new features of ES6 while maintaining the compatibility of ES5, and knowing which is the most efficient will allow these developers' new code to also run relatively fast.

1.2 Background and Motivation

ECMAScript, more commonly referred to as JavaScript (JS), is the web-standard scripting language (Crockford). Used in most websites, javascript provides advanced interactive functionality such as tooltips and animation. The functionality provided by JS to extend webpages' capabilities is incredible; However, due to poor object modeling, its name (a satire of the language Java), and numerous other "quirks", JS has a relatively poor reputation among professional developers (Crockford).

In order to improve JS and its appeal to developers, the ES6 proposal sets the following goals: improve large application support, support libraries, and make ES6 a more attractive compilation target ("ECMAScript 2015 Language Specification Final Draft" xvii). Being a better compilation target means being a better language to run code in. At the moment, JS is not the most pleasant language to use to write and run code (Crockford), and ES6 aims to fix this. However, the new features of ES6 cannot be run on an ES5 JS engine like V8 (unless the engine is modified to support ES6). Therefore, for greater compatibility, precompilers were produced to take ES6 code and recompile it with the compilation target of ES5. In summary, ES6 code run through a precompiler becomes potentially slower but far more compatible ES5 code. Given that the ES5 code produced is machine-generated, it can have various levels of efficiency. The purpose of this experiment is to test those levels, and to determine which precompiler produces the fastest code.

1.3 Features to be tested

The first feature integrated into the ES6 code to be tested for precompilation efficiency was arrow functions. Resembling the notation used in lamda calculus, arrow functions provide a more compact method of expressing and defining functions in a way similar to

C#, CoffeeScript, and other popular languages (Hoban). Given that they are processed almost exactly the same as traditional functions, efficiency of execution should not be too poorly affected by precompilation to ES5, and the resulting code should be relatively simple ("ECMAScript 2015 Language Specification Final Draft" 86, 249).

Another feature which was tested was destructuring syntax. Destructuring syntax is a compact way of assigning values to sets of variables without breaking apart and then re-combining those sets (Hoban). This feature was used as something of a wildcard, as I couldn't determine the ease of efficient compilation by reviewing the ECMAScript specification. As part of this test, the new "spread" (...) operator was also used, which does something similar in expanding data into a different form that would normally take recombination (Hoban).

Lastly, let and const declarations were tested for efficiency. let and const declarations are new methods of defining variables in ES6. let declarations provide more variable security (that is, greater restrictions on access and modification of variables) by restricting variable access to the scope of the block in which the variable is declared (Hoban). const operates in much the same way, with the additional restriction that a value assigned by const is immutable, or constant, meaning it can't be overwritten later (Hoban). The specifics of the functionality of let and const are not really necessary for this experiment, however, just the understanding that ES6 let and const statements work only slightly differently than ES5 variable declarations, yet still differently enough that extra code is necessary for correct functionality.

1.4 Hypothesis

Due to its usage in performance-intensive web technology-based applications such as Flipboard, Atom, Netflix, Cloudflare, and Yahoo!, I predicted that Babel will produce the fastest code (Babel). As further support of its more advanced development, Babel supports more ES6 features (kangax).

2 Methodology

2.1 Software Versions and Testing Environment

- Windows 8.1 64-bit Build 9600
- 12GB RAM

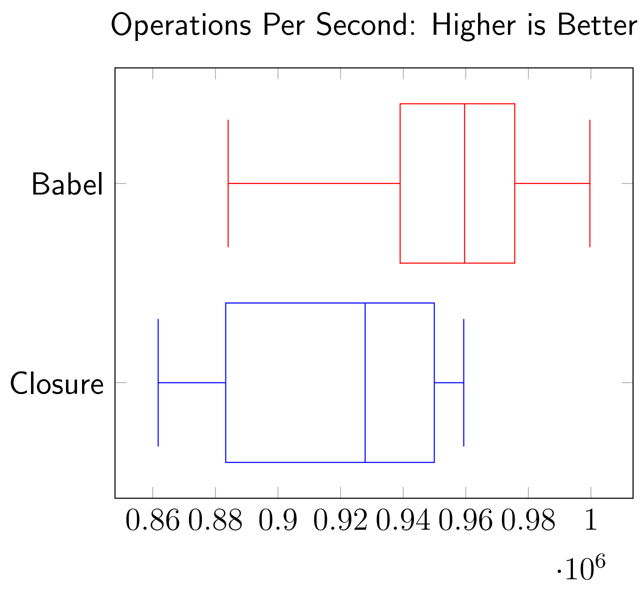
- 14GB virtual memory
- Intel i7-4710HQ
- Google Chrome 32-bit Official Build 42.0.2311.135 with flags –javascript-harmony –manual-enhanced-bookmarks
- jsPerf (http://jsperf.com/)
- Babel v5.2.15 (http://babeljs.io/)
- Closure v20150505 (https://github.com/google/closure-compiler/)

2.2 Procedures

1. The softwares were downloaded and installed
2. The code (Appendix A) was added to the file `in.js`
3. The code (Appendix A) was run through closure using the command `java -jar compiler.jar in.js --language_in=ECMASCRIPT6 --language_out=ES5 --js_output_file=closureout.js`
4. The code (Appendix A) was run through babel using the command `babel in.js -o babelout.js`
5. The contents of closureout.js and babelout.js were copied into the "Code snippet 1" and "Code snippet 2" fields of jsPerf, respectively
6. The jsPerf page was generated and assigned the url http://jsperf.com/es6testing
7. The jsPerf tests were run
8. The operations per second data were recorded
9. Steps 7-8 were repeated 14 more times

3 Results and Discussion

3.1 Data



Trial	Closure (ops/sec)	Babel (ops/sec)
1	961,760	964,424
2	948,241	982,245
3	959,354	983,706
4	957,510	984,387
5	950,520	966,674
6	942,229	959,627
7	928,698	949,760
8	925,528	969,058
9	927,008	938,628
10	880,245	939,446
11	892,422	946,013
12	957,849	999,665
13	868,791	908,324
14	873,249	900,882
15	861,712	884,077

3.2 Discussion and Possibilities for Further Research

As indicated by the box plots, Babel was clearly faster. This finding confirms my hypothesis. Interestingly, the maximum number of operations (ops) per second reached by Closure (959354) wasn't even as much as the median number of ops per second performed by Babel (959,627). This is strange, given that Closure is specifically designed to write efficient code (Google). Only as an additional feature is it a ES6 precompiler. On the other hand, for the same reason it may make more sense that Babel, designed as an ES6 precompiler, is more efficient when compiling from ES6 (Babel). In any case, further research could be performed into the specific functionality of Babel and Closure, as well as possibly other pre- and transpilers. This would provide a more deductive approach to figuring out what precompiler is the most efficient. One notable discrepancy is Babel's inclusion of the strict mode declaration `"use strict";` which may affect performance in the V8 engine. However, based on the results of this study, it is advisable to use Babel over Closure for the purposes of ES6 precompilation.

Works Cited

Babel. “Babel: The compiler for writing next generation JavaScript”. (10 May 2015). Web.

Crockford, Douglas. “Javascript: The World’s Most Misunderstood Programming Language Has Become the World’s Most Popular Programming Language”. (Mar. 2008). Web.

“ECMAScript 2015 Language Specification Final Draft”. (Apr. 2015). Web.

Google. “Closure Compiler”. *Google Developers* (1 May 2015). Web.

Hoban, Luke. “es6features: Overview of ECMAScript 6 features”. *GitHub* (18 Apr. 2015). Web.

kangax. “ECMAScript 6 compatibility table”. *GitHub* (10 May 2015). Web.

StatCounter. “Top 5 Desktop, Tablet Console Browsers from Apr 2014 to Apr 2015”. (Apr. 2015). Web.

Appendix A   Sample ES6 Code

```
let [p, q] = [0, 1];//let declaration
function nextFibo() {
        [p, q] = [q, p + q];
        return q;
}
let fibs = [];
while (true){
        let n = nextFibo();
    if (n > 10000)
        break;
    fibs.push(n)
}
let [fib1, , fib3, ...therest] = fibs; //deconstructor and
   spread
let halfTheRest = therest.map(x => x/2); //arrow function
```

Appendix B Closure Compiled ES5 Code

```
var $jscomp$destructuring$var0=[0,1],p=
   $jscomp$destructuring$var0[0],q=$jscomp$destructuring$var0
   [1];function nextFibo(){var a=[q,p+q];p=a[0];return q=a
   [1]}for(var fibs=[];;){var n=nextFibo();if(1E4<n)break;
   fibs.push(n)}var $jscomp$destructuring$var2=fibs,fib1=
   $jscomp$destructuring$var2[0],fib3=
   $jscomp$destructuring$var2[2],therest=[].slice.call(
   $jscomp$destructuring$var2,3),halfTheRest=therest.map(
   function(a){return a/2});
```

Appendix C Babel Compiled ES5 Code

```
"use strict";

var p = 0;
var q = 1;
//let declaration
function nextFibo() {
    var _ref = [q, p + q];
    p = _ref[0];
    q = _ref[1];

    return q;
}
var fibs = [];
while (true) {
    var n = nextFibo();
    if (n > 10000) break;
    fibs.push(n);
}
var fib1 = fibs[0];
var fib3 = fibs[2];
var therest = fibs.slice(3);
//deconstructor and spread
var halfTheRest = therest.map(function (x) {
    return x / 2;
}); //arrow function
```