« On joining Microsoft Edge and moving to Seattle

Progressive enhancement isn't dead, but it smells funny »

15 UG

The cost of small modules

Posted August 15, 2016 by Nolan Lawson in performance, Web. **79 Comments**

Update (30 Oct 2016): *since I wrote this post, a bug was found in the benchmark which caused Rollup to appear slightly better than it would otherwise. However, the overall results are not substantially different (Rollup still beats Browserify and Webpack, although it's not quite as good as Closure anymore), so I've merely updated the charts and tables. Additionally, the benchmark now includes the RequireJS and RequireJS Almond bundlers, so those have been added as well. To see the original blog post without these edits, check out this archived version.*

Update (21 May 2018): *This blog post analyzed older versions of Webpack, Browserify, and other module bundlers. Later versions of these bundlers added support for features like module concatenation and flat packing, which address most of the concerns raised in this blog post. You can get an idea for the performance improvement from these methods in these PRs.*

About a year ago I was refactoring a large JavaScript codebase into smaller modules, when I discovered a depressing fact about Browserify and Webpack:

> "The more I modularize my code, the bigger it gets. 😲"
> – Nolan Lawson

Later on, Sam Saccone published some excellent research on Tumblr and Imgur's page load performance, in which he noted:

> "Over 400ms is being spent simply walking the Browserify tree."
> – Sam Saccone

In this post, I'd like to demonstrate that small modules can have a surprisingly high performance cost depending on your choice of bundler and module system. Furthermore, I'll explain why this applies not only to the modules in your own codebase, but also to the modules *within dependencies*, which is a rarely-discussed aspect of the cost of third-party code.

Web perf 101

The more JavaScript included on a page, the slower that page tends to be. Large JavaScript bundles cause the browser to spend more time downloading, parsing, and executing the script, all of which lead to slower load times.

Even when breaking up the code into multiple bundles – Webpack code splitting, Browserify factor bundles, etc. – the cost is merely delayed until later in the page lifecycle. Sooner or later, the JavaScript piper must be paid.

Furthermore, because JavaScript is a dynamic language, and because the prevailing CommonJS module system is also dynamic, it's fiendishly difficult to extract unused code from the final payload that gets shipped to users. You might only need jQuery's `$.ajax`, but by including jQuery, you pay the cost of the entire library.

The JavaScript community has responded to this problem by advocating the use of small modules Small modules have a lot of aesthetic and practical benefits – easier to maintain, easier to comprehend, easier to plug together – but they also solve the jQuery problem by promoting the inclusion of small bits of functionality rather than big "kitchen sink" libraries.

So in the "small modules" world, instead of doing:

```
var _ = require('lodash')
_.uniq([1,2,2,3])
```

You might do:

Recent Posts

- Should computers serve humans, or should humans serve computers?
- Introducing Pinafore for Mastodon
- Smaller Lodash bundles with Webpack and Babel
- Decentralized identity and decentralized social networks
- 2017 book review

About Me

Hi, I'm Nolan. I help build the web at Microsoft. Opinions expressed in this blog are mine and frequently wrong.

Archives

```
var uniq = require('lodash.uniq')
uniq([1,2,2,3])
```

Rich Harris has already articulated why the “small modules” pattern is inherently beginner-unfriendly, even though it tends to make life easier for library maintainers. However, there’s also a hidden performance cost to small modules that I don’t think has been adequately explored.

Packages vs modules

It’s important to note that, when I say “modules,” I’m not talking about “packages” in the npm sense. When you install a package from npm, it might only expose a single module in its public API, but under the hood it could actually be a conglomeration of many modules.

For instance, consider a package like is-array. It has no dependencies and only contains one JavaScript file, so it has one module. Simple enough.

Now consider a slightly more complex package like once, which has exactly one dependency: wrappy. Both packages contain one module, so the total module count is 2. So far, so good.

Now let’s consider a more deceptive example: qs. Since it has zero dependencies, you might assume it only has one module. But in fact, it has four!

You can confirm this by using a tool I wrote called browserify-count-modules, which simply counts the total number of modules in a Browserify bundle:

```
$ npm install qs
$ browserify node_modules/qs | browserify-count-modules
4
```

What’s going on here? Well, if you look at the source for qs, you’ll see that it contains four JavaScript files, representing four JavaScript modules which are ultimately included in the Browserify bundle.

This means that a given *package* can actually contain one or more *modules*. These modules can also depend on other packages, which might bring in their own packages and modules. The only thing you can be sure of is that each package contains *at least* one module.

Module bloat

How many modules are in a typical web application? Well, I ran `browserify-count-modules` on a few popular Browserify-using sites, and came up with these numbers:

- requirebin.com: 91 modules
- keybase.io: 365 modules
- m.reddit.com: 1050 modules
- Apple.com: 1060 modules (*Added. Thanks, Max!*)

For the record, my own Pokedex.org (the largest open-source site I’ve built) contains 311 modules across four bundle files.

Ignoring for a moment the raw size of those JavaScript bundles, I think it’s interesting to explore the cost of the *number of modules* themselves. Sam Saccone has already blown this story wide open in “The cost of transpiling es2015 in 2016”, but I don’t think his findings have gotten nearly enough press, so let’s dig a little deeper.

Benchmark time!

I put together a small benchmark that constructs a JavaScript module importing 100, 1000, and 5000 other modules, each of which merely exports a number. The parent module just sums the numbers together and logs the result:

```
// index.js
var total = 0
total += require('./module_0')
total += require('./module_1')
total += require('./module_2')
// etc.
console.log(total)
```

```
// module_0.js
module.exports = 0
```

```
// module_1.js
module.exports = 1
```

(And so on.)

I tested five bundling methods: Browserify, Browserify with the bundle-collapser plugin, Webpack, Rollup, and Closure Compiler. For Rollup and Closure Compiler I used ES6 modules, whereas for Browserify and Webpack I used CommonJS, so as not to unfairly disadvantage them (since they would need a transpiler like Babel, which adds its own overhead).

In order to best simulate a production environment, I used Uglify with the `--mangle` and `--compress` settings for all bundles, and served them gzipped over HTTPS using GitHub

- May 2018 (1)
- April 2018 (1)
- March 2018 (1)
- January 2018 (1)
- December 2017 (1)
- November 2017 (2)
- October 2017 (1)
- August 2017 (1)
- May 2017 (1)
- March 2017 (1)
- January 2017 (1)
- October 2016 (1)
- August 2016 (1)
- June 2016 (1)
- April 2016 (1)
- February 2016 (2)
- December 2015 (1)
- October 2015 (1)
- September 2015 (1)
- July 2015 (1)
- June 2015 (2)
- October 2014 (1)
- September 2014 (1)
- April 2014 (1)
- March 2014 (1)
- December 2013 (2)
- November 2013 (3)
- August 2013 (1)
- May 2013 (3)
- January 2013 (1)

Tags

alogcat android android market apple app tracker blobs boost bootstrap bug reports bundles catlog chord reader code contacts continuous integration copyright cordova couch apps couchdb couchdroid databases developers development grails html5 indexeddb information retrieval japanese name converter javascript jenkins keepscore listview localstorage logcat logviewer lucene modules nginx nlp node nodejs npm offline-first open source passwords performance pokedroid pouchdb pouchdroid progressive enhancement pwa query expansion relatedness calculator relatedness coefficient s3 safari sectioned listview security semver social media socket.io software development solr sqlite supersaiyanscrollview synonyms ui design ultimate crossword w3c webapp webapps web

Pages. For each bundle, I downloaded and executed it 15 times and took the median, noting the (uncached) load time and execution time using `performance.now()`.

Bundle sizes

Before we get into the benchmark results, it's worth taking a look at the bundle files themselves. Here are the byte sizes (minified but ungzipped) for each bundle (chart view):

	100 modules	1000 modules	5000 modules
browserify	7982	79987	419985
browserify-collapsed	5786	57991	309982
webpack	3955	39057	203054
rollup	1265	13865	81851
closure	758	7958	43955
rjs	29234	136338	628347
rjs-almond	14509	121612	613622

And the minified+gzipped sizes (chart view):

	100 modules	1000 modules	5000 modules
browserify	1650	13779	63554
browserify-collapsed	1464	11837	55536
webpack	688	4850	24635
rollup	629	4604	22389
closure	302	2140	11807
rjs	7940	19017	62674
rjs-almond	2732	13187	56135

What stands out is that the Browserify and Webpack versions are much larger than the Rollup and Closure Compiler versions (***update:*** *especially before gzipping, which still matters since that's what the browser executes*). If you take a look at the code inside the bundles, it becomes clear why.

The way Browserify and Webpack work is by isolating each module into its own function scope, and then declaring a top-level runtime loader that locates the proper module whenever `require()` is called. Here's what our Browserify bundle looks like:

```
(function e(t,n,r){function s(o,u){if(!n[o]){if(!t[o]){var a=typeof require=="function"&&require;if(!u&&
module.exports = 0
},{}],2:[function(require,module,exports){
module.exports = 1
},{}],3:[function(require,module,exports){
module.exports = 10
},{}],4:[function(require,module,exports){
module.exports = 100
// etc.
```

Whereas the Rollup and Closure bundles look more like what you might hand-author if you were just writing one big module. Here's Rollup:

```
(function () {
	'use strict';
	var module_0 = 0
	var module_1 = 1
	// ...
	total += module_0
	total += module_1
	// etc.
```

The important thing to notice is that every module in Webpack and Browserify gets its own function scope, and is loaded at runtime when `require()`d from the main script. Rollup and Closure Compiler, on the other hand, just hoist everything into a single function scope (creating variables and namespacing them as necessary).

If you understand the inherent cost of functions-within-functions in JavaScript, and of looking up a value in an associative array, then you'll be in a good position to understand the following benchmark results.

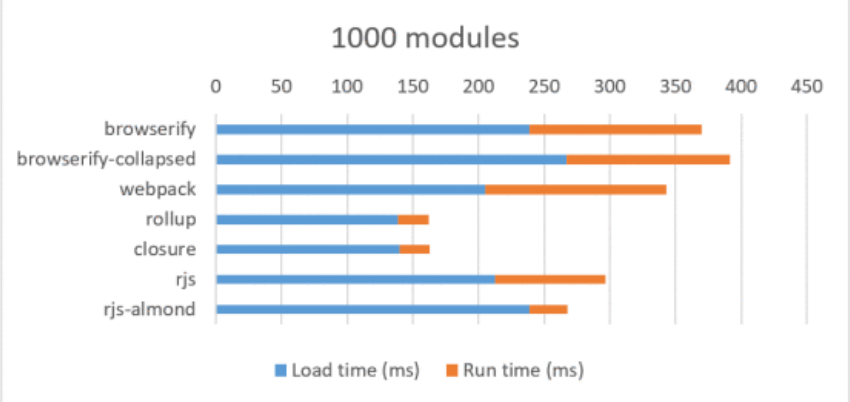
platform web sockets websql web workers

Results

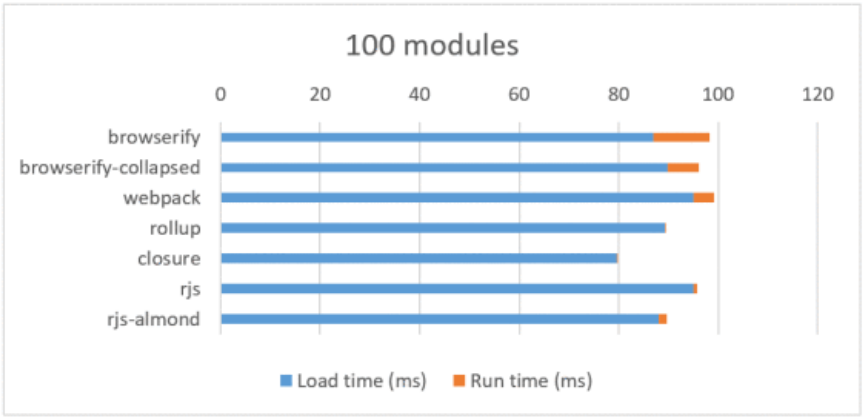
Update: *as noted above, results have been re-run with corrections and the addition of the r.js and r.js Almond bundlers. For the full tabular data, see this gist.*

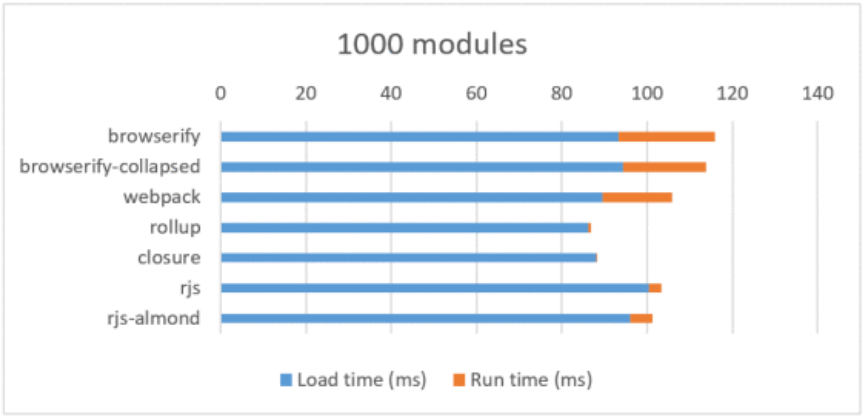
I ran this benchmark on a Nexus 5 with Android 5.1.1 and Chrome 52 (to represent a low- to mid-range device) as well as an iPod Touch 6th generation running iOS 9 (to represent a high-end device).

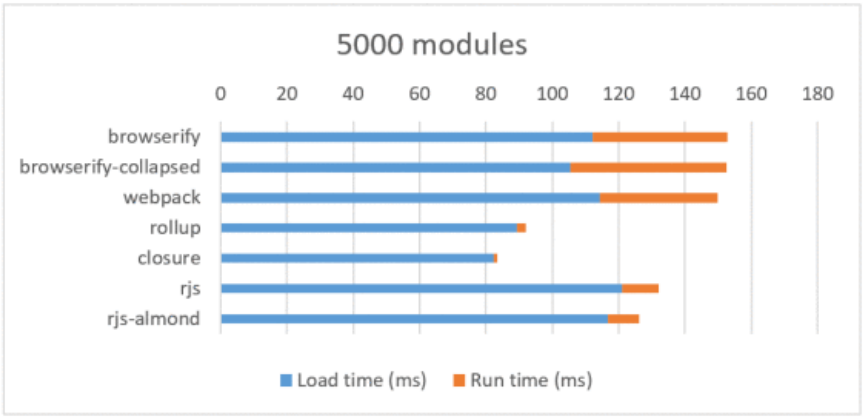
Here are the results for the Nexus 5:



And here are the results for the iPod Touch:



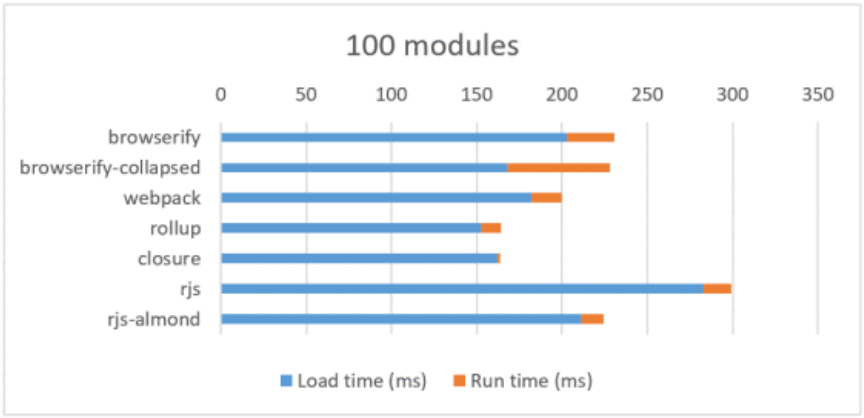


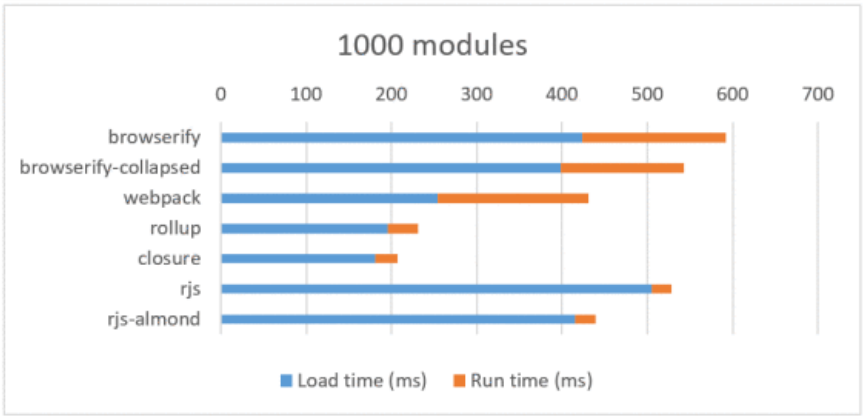


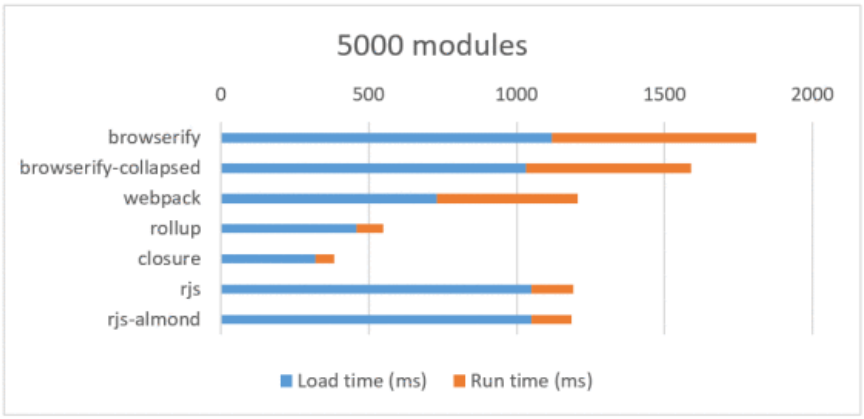
At 100 modules, the variance between all the bundlers is pretty negligible, but once we get up to 1000 or 5000 modules, the difference becomes severe. The iPod Touch is hurt the least by the choice of bundler, but the Nexus 5, being an aging Android phone, suffers a lot under Browserify and Webpack.

I also find it interesting that both Rollup and Closure's execution cost is essentially free for the iPod, regardless of the number of modules. And in the case of the Nexus 5, the runtime costs aren't free, but they're still much cheaper for Rollup/Closure than for Browserify/Webpack, the latter of which chew up the main thread for several frames if not hundreds of milliseconds, meaning that the UI is frozen just waiting for the module loader to finish running.

Note that both of these tests were run on a fast Gigabit connection, so in terms of network costs, it's really a best-case scenario. Using the Chrome Dev Tools, we can manually throttle that Nexus 5 down to 3G and see the impact:







Once we take slow networks into account, the difference between Browserify/Webpack and Rollup/Closure is even more stark. In the case of 1000 modules (which is close to Reddit's count of 1050), Browserify takes about 400 milliseconds longer than Rollup. And that 400ms is no small potatoes, since Google and Bing have both noted that sub-second delays have an appreciable impact on user engagement.

One thing to note is that this benchmark doesn't measure the precise execution cost of 100, 1000, or 5000 modules *per se*, since that will depend on your usage of `require()`. Inside of these bundles, I'm calling `require()` once per module, but if you are calling `require()` multiple times per module (which is the norm in most codebases) or if you are calling `require()` multiple times on-the-fly (i.e. `require()` within a sub-function), then you could see severe performance degradations.

Reddit's mobile site is a good example of this. Even though they have 1050 modules, I clocked their real-world Browserify execution time as much worse than the "1000 modules" benchmark. When profiling on that same Nexus 5 running Chrome, I measured 2.14 seconds for Reddit's Browserify `require()` function, and 197 milliseconds for the equivalent function in the "1000 modules" script. (In desktop Chrome on an i7 Surface Book, I also measured it at 559ms vs 37ms, which is pretty astonishing given we're talking *desktop*.)

This suggests that it may be worthwhile to run the benchmark again with multiple `require()`s per module, although in my opinion it wouldn't be a fair fight for Browserify/Webpack, since Rollup/Closure both resolve duplicate ES6 imports into a single hoisted variable declaration, and it's also impossible to `import` from anywhere but the top-level scope. So in essence, the cost of a single `import` for Rollup/Closure is the same as the cost of *n* `import`s, whereas for Browserify/Webpack, the execution cost will increase linearly with *n* `require()`s.

For the purposes of this analysis, though, I think it's best to just assume that the number of

modules is only a *lower* bound for the performance hit you might feel. In reality, the "5000 modules" benchmark may be a better yardstick for "5000 `require()` calls."

Conclusions

First off, the `bundle-collapser` plugin seems to be a valuable addition to Browserify. If you're not using it in production, then your bundle will be a bit larger and slower than it would be otherwise (although I must admit the difference is slight). Alternatively, you could switch to Webpack and get an even faster bundle without any extra configuration. (Note that it pains me to say this, since I'm a diehard Browserify fanboy.)

However, these results clearly show that Webpack and Browserify both underperform compared to Rollup and Closure Compiler, and that the gap widens the more modules you add. Unfortunately I'm not sure Webpack 2 will solve any of these problems, because although they'll be borrowing some ideas from Rollup, they seem to be more focused on the tree-shaking aspects and not the scope-hoisting aspects. *(**Update:** a better name is "inlining," and the Webpack team is working on it.)*

Given these results, I'm surprised Closure Compiler and Rollup aren't getting much traction in the JavaScript community. I'm guessing it's due to the fact that (in the case of the former) it has a Java dependency, and (in the case of the latter) it's still fairly immature and doesn't quite work out-of-the-box yet (see Calvin's Metcalf's comments for a good summary).

Even without the average JavaScript developer jumping on the Rollup/Closure bandwagon, though, I think npm package authors are already in a good position to help solve this problem. If you `npm install lodash`, you'll notice that the main export is one giant JavaScript module, rather than what you might expect given Lodash's hyper-modular nature (`require('lodash/uniq')`, `require('lodash.uniq')`, etc.). For PouchDB, we made a similar decision to use Rollup as a prepublish step, which produces the smallest possible bundle in a way that's invisible to users.

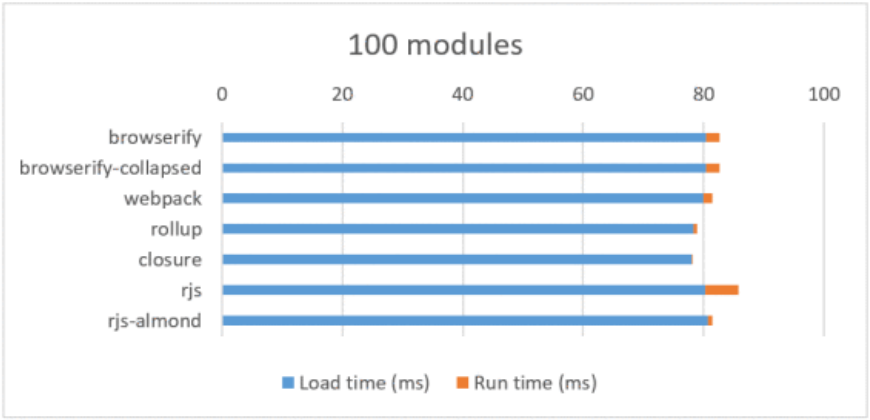
I also created rollupify to try to make this pattern a bit easier to just drop-in to existing Browserify projects. The basic idea is to use `import`s and `export`s within your own project (cjs-to-es6 can help migrate), and then use `require()` for third-party packages. That way, you still have all the benefits of modularity within your own codebase, while exposing more-or-less one big module to your users. Unfortunately, you still pay the costs for third-party modules, but I've found that this is a good compromise given the current state of the npm ecosystem.

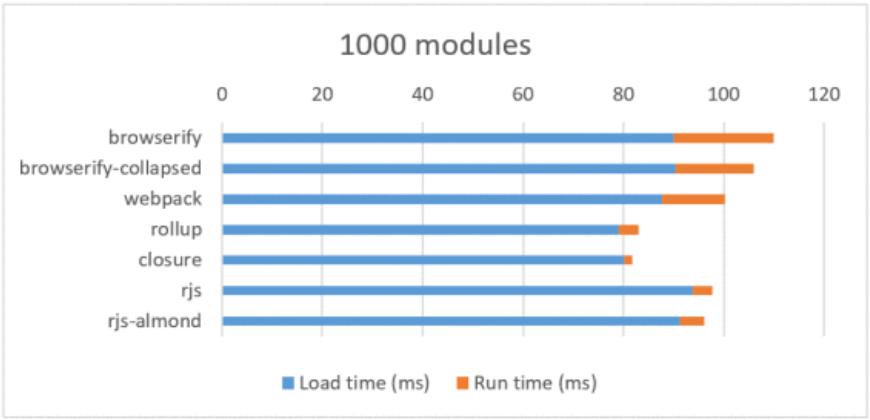
So there you have it: one horse-sized JavaScript duck is faster than a hundred duck-sized JavaScript horses. Despite this fact, though, I hope that our community will eventually realize the pickle we're in – advocating for a "small modules" philosophy that's good for developers but bad for users – and improve our tools, so that we can have the best of both worlds.

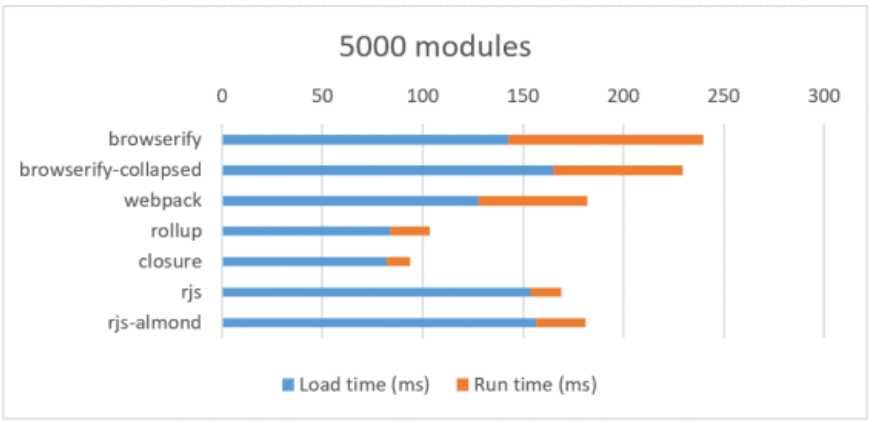
Bonus round! Three desktop browsers

Normally I like to run performance tests on mobile devices, since that's where you see the clearest differences. But out of curiosity, I also ran this benchmark on Chrome 54, Edge 14, and Firefox 48 on an i5 Surface Book using Windows 10 RS1. Here are the results:

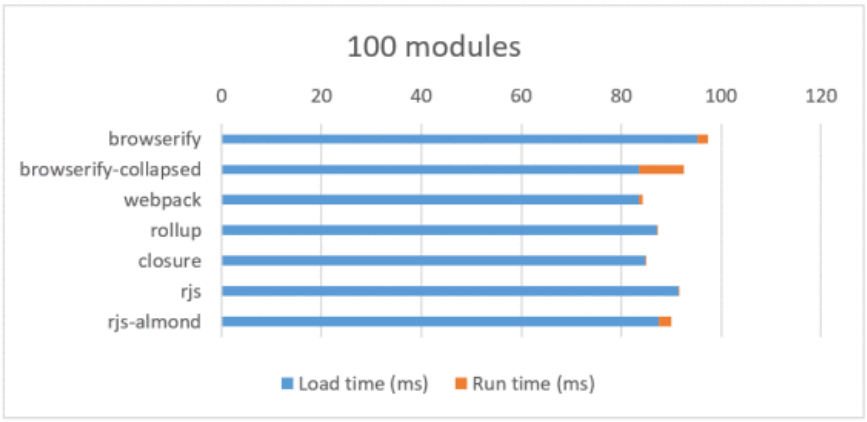
Chrome 54

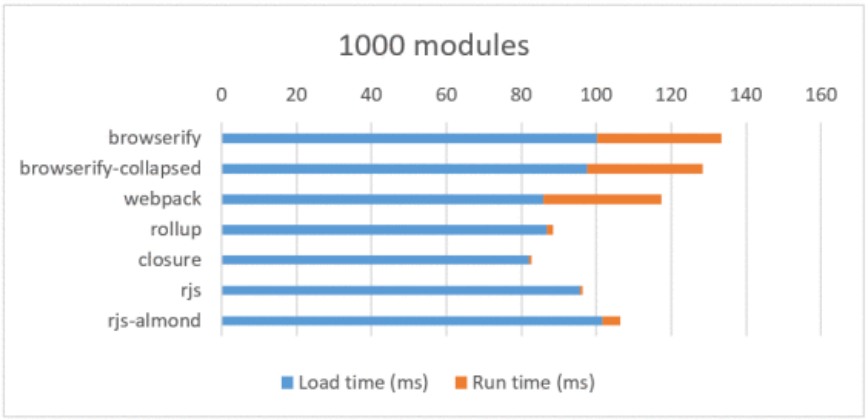




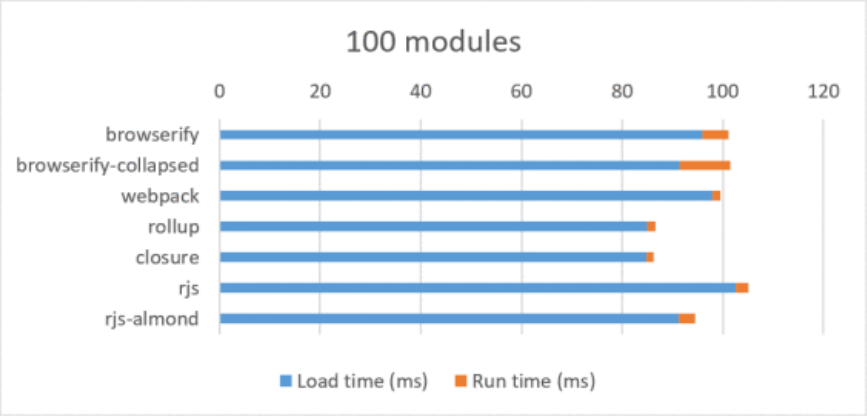


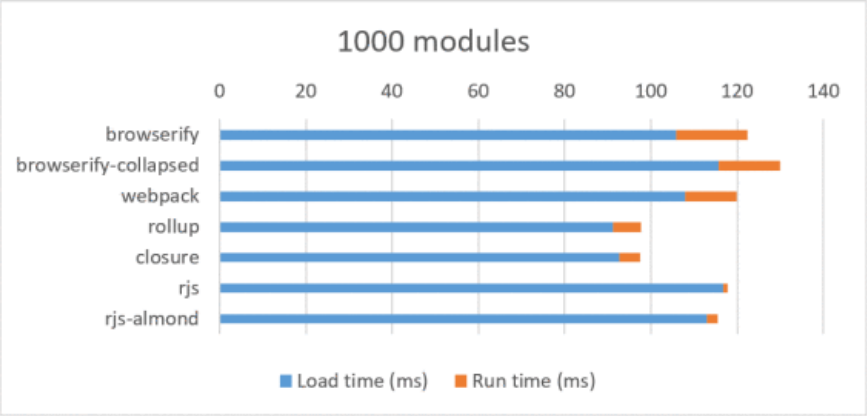
Edge 14 (tabular results)





Firefox 48 (tabular results)





The only interesting tidbits I'll call out in these results are:

1. `bundle-collapser` is definitely not a slam-dunk in all cases.
2. The ratio of network-to-execution time is always extremely high for Rollup and Closure; their runtime costs are basically zilch. ChakraCore and SpiderMonkey eat them up for breakfast, and V8 is not far behind.

This latter point could be extremely important if your JavaScript is largely lazy-loaded, because if you can afford to wait on the network, then using Rollup and Closure will have the additional benefit of not clogging up the UI thread, i.e. they'll introduce less jank than Browserify or Webpack.

Update: in response to this post, JDD has opened an issue on Webpack. There's also one on Browserify.

Update 2: Ryan Fitzer has generously added RequireJS and RequireJS with Almond to the benchmark, both of which use AMD instead of CommonJS or ES6.

Testing shows that RequireJS has the largest bundle sizes but surprisingly its runtime costs are very close to Rollup and Closure. (See updated results above for details.)

Update 3: I wrote optimize-js, which alleviates some of the performance costs of parsing functions-within-functions.

Related

A brief and incomplete history of JavaScript bundlers
In "Web"

How to write a JavaScript package for both Node and the browser
In "Web"

The struggles of publishing a JavaScript library
In "Webapps"

79 responses to this post.

Posted by kimbynd on August 15, 2016 at 10:57 AM

It'd be interesting to see a comparison with JSPM/System JS, and accounting for caching of modules, saving multiple network requests

Reply

Posted by Nolan Lawson on August 21, 2016 at 8:06 AM

JSPM/SystemJS **uses Rollup under the hood**. As for caching of modules, these are single bundles, so there's only one network request.

Reply

Posted by jetpl on August 16, 2016 at 2:33 AM

Agree, would be great if you could prepare a comparison based also on JSPM/System JS.
Thanks for a great article.

Reply

Posted by Nolan Lawson on August 17, 2016 at 7:34 AM

Seems JSPM/SystemJS **uses Rollup under the hood**.

Reply

Posted by dino on September 3, 2016 at 4:20 PM

JSPM uses rollup for any ES modules, your own package and maybe some 3rd party, but most 3rd party modules are imported in commonJS or UMD module formats.

JSPM build command will list the module which had been optimised with rollup.

JSPM offers can override a package definition (which where to find the main module where is source directory, etc.) to use its ES module if the npm package include them.

Posted by tobielangel on August 16, 2016 at 3:36 AM

Yes. Scope-hoisting is critical. It's a complex but doable AST transform. It's sort of disappointing (but not very surprising) to see Google nailing this close to a decade ago, and very little happening in the JS community on that topic since.

Reply

Posted by Nolan Lawson on August 17, 2016 at 7:33 AM

I think the reason is that priority #1 with Webpack and (especially) Browserify was compatibility with the Node ecosystem. They needed to be able to use a large

percentage of npm's 300,000 packages in order to gain traction.

That meant implementing `require()` to a T despite all its faults (dynamic, not-top-scoped, try/catchable), plus implementing a heck of a lot of Node-isms – **calvinmetcalf/immediate** and **feross/buffer** are a testament to how complex it is to migrate Node APIs like `process.nextTick` and `Buffer` to the browser.

But now that we've got a large number of npm's 300,000 modules available in the browser, it's time to optimize. Step #1 unfortunately may be to try to convince popular package authors to switch from CommonJS to ES6 modules, or maybe to write a Rollup-like optimizer for CommonJS that bails out with an unoptimizable pattern is used.

Reply

Posted by tobielangel on August 27, 2016 at 2:47 AM

Oh that's interesting. I don't recall constructs outside of dynamic module names that were common and that would have prevented module hoisting. But that was nearly 5 years ago and my memory's kind of fuzzy at this point. iirc, I was also hawkish about preventing features that might have made the system less optimizable for perf.

Posted by Vasco on August 18, 2016 at 10:29 AM

> *see Google nailing this close to a decade ago*

Do you mean that GWT supported this ?

Reply

Posted by Nolan Lawson on August 18, 2016 at 4:45 PM

Think he meant Google Closure Compiler.

Posted by tobielangel on August 27, 2016 at 2:48 AM

What nolan said.

Posted by garbas on August 16, 2016 at 6:49 AM

would this be possible to extend to Elm?

Reply

Posted by timmywil on August 16, 2016 at 7:16 AM

This is neither here nor there, but you can use jQuery.ajax on its own either through the use of AMD (or a module bundler that can consume AMD).

Reply

Posted by timmywil on August 16, 2016 at 7:21 AM

Oh, besides AMD, you can use the custom build system and just include ajax.

Reply

Posted by CouchDB Weekly News, August 18, 2016 – CouchDB Blog on August 18, 2016 at 9:06 AM

[…] The cost of small modules, Nolan […]

Reply

Posted by dtothefp on August 18, 2016 at 11:37 AM

I agree that Rollup style bundles are much more effective by throwing all your modules into a single closure scope. Although we encountered many problems when implementing Rollup in anything more than a trivial project. To get it to play well with non es6 dependencies requires confusing config and issues on their GH suggest that it won't even work for React/Redux projects **https://github.com/rollup/rollup-plugin-commonjs/issues/29**. Also from placing issues on multiple plugins and receiving little to no response without these issues ever being closed was frustrating and suggests the project is not actively maintained.

If you have some magic bullets for Rollup it would be great to learn them but otherwise it doesn't seem mature enough

Reply

Posted by Hanter on August 18, 2016 at 10:22 PM

I stoped reading at "The way Browserify and Webpack work is by isolating each module into its own function scope". This is true for webpack in dev-mode. Have you ever tried "webpack -p" for production mode?
https://webpack.github.io/docs/cli.html

Reply

Posted by Sean Vieira on August 19, 2016 at 2:17 PM

Does the same individual module wrapping, just orders them so that more commonly required ones have smaller numbers.

Reply

Posted by Nolan Lawson on August 19, 2016 at 6:06 PM

Yup, I didn't use `--production`, but in this case it shouldn't make a difference because each module is used exactly once. I ran it just now and verified that the only difference is that `--production` orders the `require()`s based on numeric order (because that's the order they're used) rather than the default lexicographic order (i.e. 1,2,3,4,5,… instead of 1,10,2,20,3,30,…).

The bundle size will end up being the same, and the different ordering may have a teensy impact on runtime perf depending on how the underlying hashmap optimizes access order, but ought to be largely the same (although of course, it's worth testing. :)).

Posted by Nolan Lawson on August 20, 2016 at 2:46 PM

I've added `webpack -p` to the build script, don't see much change in the results.

Posted by foljs on August 21, 2016 at 3:23 PM

@Hanter >I stoped reading at "The way Browserify and Webpack work is by isolating each module into its own function scope".

I stopped reading when I read your rude comment about when you stopped reading. Someone takes the time to write a large post on an important performance issue, and the only think you can contribute is a knee jerk pedantic correction, and for something that's inconsequential at that.

Besides, if you haven't "stopped reading" articles before their end, you might have learned by now that an article can contain one or more factual errors or misconceptions, and still have valuable insights and information.

Reply

Posted by Web Development Reading List #150: Less Code, GitHub's Security, And The Morals Of Science – Smashing Magazine on August 19, 2016 at 2:36 AM

[…] Lawson wrote about the cost of small modules14, analyzing how much code is used when you build your codebase with a lot of small modules. The […]

Reply

Posted by Web Development Reading List #150: Less Code, GitHub's Security, And The Morals Of Science - American Fido on August 19, 2016 at 6:30 AM

[…] Lawson wrote about the cost of small modules14, analyzing how much code is used when you build your codebase with a lot of small modules. The […]

Reply

Posted by #WAT-Up7.0 on August 19, 2016 at 8:50 AM

[…] There are many tools that will compile your es2015 code into es5 or lower, but are they all performance-friendly? This github repo experiments with a variety of tools to test overall compile size, runtime, and JS compile and execution time on page load. The biggest take away? "When choosing your compile stack you should be aware that tools that perform tree shaking and topological sorts of your code dependencies will result in smaller code bundles and faster js execution times." If you want more proof, check out Nolan Lawson's take on the issue. […]

Reply

Posted by Web Development Reading List #150: Less Code, GitHub's Security, And The Morals Of Science on August 19, 2016 at 12:11 PM

[…] Lawson wrote about the cost of small modules14, analyzing how much code is used when you build your codebase with a lot of small modules. The […]

Reply

Posted by Weekly Links #28 | Useful Links For Developers on August 20, 2016 at 2:18 PM

[…] The cost of small modules […]

Reply

Posted by Joe Zimmerman on August 20, 2016 at 9:10 PM

Yea, saw Rollup a while back but it seemed very immature so I didn't think much of it. Advertised the tree shaking a lot, which I figured would either not work as well as advertised or would get added to the big name bundlers. The single scope thing also scared the heck out of me, so I decided ignore it. I'm hoping ES2015 will be fully ready to use in browsers soon plus HTTP2 so we can just forget this whole mess.

Reply

Posted by Nolan Lawson on August 21, 2016 at 8:11 AM

I suspect bundling will always be faster than native ES6 modules in the browser, even in an H2 world. Khan Engineering has **a blog post** on this, but what it comes down to is the benefits of gzip operating on one big bundle.

Rollup's tree-shaking and scope-hoisting work great in my experience; where it gets tricky is all the hard edge cases that Browserify/Webpack have already solved, such as `global`, `process.nextTick()`, `Buffer`, etc. Rollup has plugins, but not all of them are complete, and there's a lot of configuration.

Reply

Posted by Collective #239 - ADD_dm on August 21, 2016 at 1:19 AM

[…] The cost of small modules […]

Reply

Posted by Дайджест свежих материалов из мира фронтенда за последнюю неделю №224 (15 — 21 августа 2016) - itfm.pro on August 21, 2016 at 2:51 PM

[…] Скорость загрузки и выполнения маленьких модулей (The cost of small modules) […]

Reply

Posted by □□□□□□□□-20160822-IT□□□ on August 21, 2016 at 8:51 PM

[…] cost of small modules **https://nolanlawson.com/2016/08/15/the-cost-of-small-modules/** In this post, I'd like to demonstrate that small modules can have a surprisingly high performance […]

Reply

Posted by Collective #239 | Latest News and RSS Feeds on August 22, 2016 at 2:39 AM

[…] The cost of small modules […]

Reply

Posted by tkane2000 on August 22, 2016 at 6:01 AM

Can we assume a direct correlation between bundle size and load time or am I missing something here?

Reply

Posted by Nolan Lawson on August 22, 2016 at 6:20 PM

Yes, that's a pretty fair assumption. Although "load" in this case also includes script parsing time, which will be longer for more complex scripts (regardless of length).

Reply

Posted by wmadden on August 23, 2016 at 12:49 AM

What about webpack/ES6 modules? You pointed out yourself that language level optimizations with CommonJS modules are next to impossible, then you pit two CommonJS compilers against ES6 module compilers. That seems a bit silly.

Anyway, is WebPack's performance really so poor in and of itself, or is it a consequence of using CommonJS modules?

Reply

Posted by Nolan Lawson on August 23, 2016 at 11:08 AM

Yes, I probably could have called this article "The Cost of CommonJS." Current Webpack plans to fix the issues I described will probably only apply to dependencies that export ES6 modules, not CJS.

OTOH Closure **claims** to be able to apply some of these optimizations to CJS, but I'm a bit skeptical because there are edge cases it undoubtedly can't handle due to CJS's dynamic nature (conditional exports/imports, try/catch imports/exports, `var mymodule = exports; mymodule.foo = 'bar';` etc.).

Reply

Posted by srcspider on August 23, 2016 at 1:29 AM

With webpack you *should* be using require.ensure and only distributing the minimal required code per page (which shouldn't really be that big). So… the so called performance comparison is a bit unfair. Yes X performs better when you have 1000, but Y prevents you from having 1000 to begin with.

Reply

Posted by Nolan Lawson on August 23, 2016 at 11:16 AM

I don't feel it's an unfair characterization. As I say in the post, `require.ensure`, code-splitting, etc. can only delay the cost of the bundled modules. And the cost still exists, even for a small number of modules.

The point of my article is that our bundlers are not free, although arguably they should be. Even with code-splitting, in the ideal case you'd split 1000 modules into 500 and 500 (for example), but within those 500-sized bundles, you wouldn't be paying an extra execution cost just for the runtime module resolution algorithm. Rollup and Closure prove this is possible.

Reply

Posted by Front-End Development (21 August) – Skokov on August 23, 2016 at 6:52 AM

[…] Tip: What Are Factory Functions in JavaScript The Marvellously Mysterious JavaScript Maybe Monad The cost of small modules Untangling Deeply-Nested Promise Chains 100+ Emulators Written in JavaScript GPU Deep Learning […]

Reply

Posted by jagretz on August 24, 2016 at 10:47 AM

Just wanted to say thank you for the awesome post! Very informative, updated with links to corresponding updates, and you've responded to many comments. Great article Nolan!

Reply

Posted by Nolan Lawson on September 23, 2016 at 2:53 PM

Thank you! :)

Reply

Posted by Chema Balsas (@jbalsas) on August 26, 2016 at 5:16 AM

Closure Compiler is now available in JS flavour at **https://github.com/google/closure-compiler-js** ;)

Reply

Posted by zaenk on September 2, 2016 at 12:19 AM

wow, this article is getting old so quickly! :D

Reply

Posted by Nolan Lawson on September 23, 2016 at 2:52 PM

It's worth testing, but my understanding is that the JS version is a port of the Java version, meaning that the output should be the same.

Posted by lr on September 3, 2016 at 11:34 AM

How about **Lasso**?

Reply

Posted by Alex on September 5, 2016 at 10:35 PM

I wonder if Brunch would do any better/worse than webpack?

Reply

Posted by Revision 276 – Große Module, kleine Module? Viel Code, wenig Code? | Working Draft on September 14, 2016 at 10:00 PM

[…] unterschiedlich viel Respekt zollen) weiter bestehen und verwendet werden, obwohl auch diese ihre ganz eigenen Probleme haben. Am Ende diskutieren wir über den Weg der Zukunft: kleine Module? Große Module mit […]

Reply

Posted by □□□□Rollup□□JavaScript - □□ - □□□□ on September 15, 2016 at 10:07 PM

[…] The cost of small modules – □□□□□□□□□Rollup□□□□□□□□□□□□Rollup□□webpack□Browserify□□ □□ […]

Reply

Posted by Aditya Vohra on September 19, 2016 at 1:50 PM

Sorry for the incredibly long comment. I might do a followup blog post if people would like a better place to discuss things.

While I greatly appreciate the research and findings in this blog post from a strictly performance perspective, I have some gripes with this statement – “Given these results, I’m surprised Closure Compiler and Rollup aren’t getting much traction in the JavaScript community.”

It’s quite possible I’m wrong about some of these (and I’d love to be corrected). Here are a few things I’ve found personally that have made it hard to work with Closure Compiler:

It’s relatively hard to integrate third party libraries

Say I want to bundle/compile third party libraries with my project’s source. Closure Compiler has some quirks (which allow it to optimize code better) that make it hard to do so. For example, Closure minifies string literal object keys and identifier object keys differently. Not a lot (if any) modern open source library authors would keep that in mind when writing libraries. This makes third party libraries incompatible with compilation by Closure in ADVANCED compilation mode (the mode one would use to achieve max compression).

http://blog.persistent.info/2015/05/teaching-closure-compiler-about-react.html talks about how a custom compiler pass had to be written to teach Closure about React constructs. And considering Closure doesn’t provide a command line option to specify command line passes, they probably forked Closure to implement a custom command line runner. I personally like to steer away from maintaining forks of giant projects (however small the differences).

So if you’re not bundling third party libraries, you’re probably going to pre-load them onto your website’s pages and use externs as described here – **https://developers.google.com/closure/compiler/docs/api-tutorial3**. The Closure folks have made externs for some common open source libraries available online – **https://github.com/google/closure-compiler/wiki/Externs-For-Common-Libraries**. The latest jQuery externs are for v1.9. jQuery is now at v3.1. The latest React externs are for v0.13.2. React is now at v15 (with a change in how they version things). And if you can’t find externs for the third party library you want to use, you’re going to have to write and maintain them yourself, and keep them up to date with updates to the library.

Devs might have to wait a fair bit between JS changes in dev

Your benchmarks don’t mention how long it takes to compile the bundles produced from 100, 1000, and 5000 modules for the various bundlers used. More importantly, you don’t talk about how long a dev would have to wait between any JS changes if they were using Closure and were compiling a giant codebase (if you’re using ES6 with Closure,

you're probably compiling your code even in dev and using sourcemaps). Closure doesn't support incremental compilation whereas Webpack and other bundlers do. Waiting more than a couple of seconds between JS changes is a red flag. With incremental building, you wait on the order of milliseconds.

Further, the more advanced the compiled bundle in its optimizations (inlining, moving code around, dead code elimination), the poorer the sourcemaps. If devs can't set breakpoints easily, that's a red flag.

Closure JS isn't exactly canonical JS

Non-ES6 Closure JS involves using Closure primitives such as goog.provide, goog.require, goog.module, etc. to declare dependencies between files/namespaces. No other compiler understands these constructs out of the box. So you're bound to the Closure Compiler as long as you continue writing JS that includes any of these constructs. Also, not a lot of people will care about any JS you open source if it uses Google Closure primitives.

If you choose to use ES6 and compile that with Closure, you're still restricted in terms of what you can use. **https://github.com/google/closure-compiler/wiki/ECMAScript6** shows what ES6 features Closure supports. The Closure folks admit that "supporting the entire ES6 specification is a non-goal for the Closure Compiler". They do support a lot, but there might come a point where you want to use some ES6 features that Closure doesn't support. Even the ES6 features that are supported have some Closure quirks (object shorthand notation, what modern linters like eslint recommend, minify differently with Closure than other compilers).

There isn't enough open source tooling around Closure Compiler

Google open sourced the compiler, but they didn't open source any of the tooling the company probably uses internally to support the usage of the compiler.

As an example, the compiler doesn't do any file discovery whatsoever. It needs to be passed in all the JS files you want it to compile as individual command line arguments. This means you need to figure out what files the compiler needs to be called with are. What's worse is that the order of the JS inputs to the Closure Compiler matters (unless you pass in certain other flags). Closure does support glob patterns as a means to specify inputs, which helps with this issue a little. But you basically need to write and maintain all the logic that passes the right arguments to Closure in the right order.

http://plovr.com/ is an open source project that can be used as a wrapper around the Closure Compiler. But that project lags behind in its updates of the bundled Closure Compiler by at least a few months, and it doesn't support the resolution of ES6 modules.

Reply

Posted by adityavohra7 on September 19, 2016 at 1:51 PM

Sorry for the incredibly long comment. I might followup with an actual blog post if people would like a better place to discuss things.

While I greatly appreciate the research and findings in this blog post from a strictly performance perspective, I have some gripes with this statement – "Given these results, I'm surprised Closure Compiler and Rollup aren't getting much traction in the JavaScript community."

It's quite possible I'm wrong about some of these (and I'd love to be corrected). Here are a few things I've found personally that have made it hard to work with Closure Compiler:

It's relatively hard to integrate third party libraries

Say I want to bundle/compile third party libraries with my project's source. Closure Compiler has some quirks (which allow it to optimize code better) that make it hard to do so. For example, Closure minifies string literal object keys and identifier object keys differently. Not a lot (if any) modern open source library authors would keep that in mind when writing libraries. This makes third party libraries incompatible with compilation by Closure in ADVANCED compilation mode (the mode one would use to achieve max compression).

http://blog.persistent.info/2015/05/teaching-closure-compiler-about-react.html talks about how a custom compiler pass had to be written to teach Closure about React constructs. And considering Closure doesn't provide a command line option to specify command line passes, they probably forked Closure to implement a custom command line runner. I personally like to steer away from maintaining forks of giant projects (however small the differences).

So if you're not bundling third party libraries, you're probably going to pre-load them onto

your website's pages and use externs as described here – **https://developers.google.com/closure/compiler/docs/api-tutorial3**. The Closure folks have made externs for some common open source libraries available online – **https://github.com/google/closure-compiler/wiki/Externs-For-Common-Libraries**. The latest jQuery externs are for v1.9. jQuery is now at v3.1. The latest React externs are for v0.13.2. React is now at v15 (with a change in how they version things). And if you can't find externs for the third party library you want to use, you're going to have to write and maintain them yourself, and keep them up to date with updates to the library.

Devs might have to wait a fair bit between JS changes in dev

Your benchmarks don't mention how long it takes to compile the bundles produced from 100, 1000, and 5000 modules for the various bundlers used. More importantly, you don't talk about how long a dev would have to wait between any JS changes if they were using Closure and were compiling a giant codebase (if you're using ES6 with Closure, you're probably compiling your code even in dev and using sourcemaps). Closure doesn't support incremental compilation whereas Webpack and other bundlers do. Waiting more than a couple of seconds between JS changes is a red flag. With incremental building, you wait on the order of milliseconds.

Further, the more advanced the compiled bundle in its optimizations (inlining, moving code around, dead code elimination), the poorer the sourcemaps. If devs can't set breakpoints easily, that's a red flag.

Closure JS isn't exactly canonical JS

Non-ES6 Closure JS involves using Closure primitives such as goog.provide, goog.require, goog.module, etc. to declare dependencies between files/namespaces. No other compiler understands these constructs out of the box. So you're bound to the Closure Compiler as long as you continue writing JS that includes any of these constructs. Also, not a lot of people will care about any JS you open source if it uses Google Closure primitives.

If you choose to use ES6 and compile that with Closure, you're still restricted in terms of what you can use. **https://github.com/google/closure-compiler/wiki/ECMAScript6** shows what ES6 features Closure supports. The Closure folks admit that "supporting the entire ES6 specification is a non-goal for the Closure Compiler". They do support a lot, but there might come a point where you want to use some ES6 features that Closure doesn't support. Even the ES6 features that are supported have some Closure quirks (object shorthand notation, what modern linters like eslint recommend, minify differently with Closure than other compilers).

There isn't enough open source tooling around Closure Compiler

Google open sourced the compiler, but they didn't open source any of the tooling the company probably uses internally to support the usage of the compiler.

As an example, the compiler doesn't do any file discovery whatsoever. It needs to be passed in all the JS files you want it to compile as individual command line arguments. This means you need to figure out what files the compiler needs to be called with are. What's worse is that the order of the JS inputs to the Closure Compiler matters (unless you pass in certain other flags). Closure does support glob patterns as a means to specify inputs, which helps with this issue a little. But you basically need to write and maintain all the logic that passes the right arguments to Closure in the right order.

http://plovr.com/ is an open source project that can be used as a wrapper around the Closure Compiler. But that project lags behind in its updates of the bundled Closure Compiler by at least a few months, and it doesn't support the resolution of ES6 modules.

Reply

Posted by adityavohra7 on September 19, 2016 at 2:01 PM

s/"And considering Closure doesn't provide a command line option to specify command line passes"/"And considering Closure doesn't provide a command line option to specify custom compiler passes"

Reply

Posted by □□ React.js □□□□ Web □□□□□□ 2 □□ - □□□□□□ - □□□□ on November 16, 2016 at 4:52 AM

[…] □□□□□□ […]

Reply

Posted by JavaScript □□□□□□ | Operations Note on November 16, 2016 at 11:49 PM

[…] □□□2016/10/30□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ Rollup □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□Rollup □□□□□ Browserify □ Webpack□□□□□□□□ Closure □□□□□□□□□□□□□□□□□□□□□□□□□□□ RequireJS □ RequireJS Almond □□ […]

Reply

Posted by [□]□□□□□□□Rollup□□JavaScript — □JSER on November 20, 2016 at 11:32 AM

[…] The cost of small modules – □□□□□□□□□Rollup□□□□□□□□□□□□□Rollup□□webpack□Browserify□□ □□ […]

Reply

Posted by Front-End Performance Checklist 2017 (PDF, Apple Pages) – Smashing Magazine on December 21, 2016 at 2:34 AM

[…] ECMAScript 2015 modules into one big CommonJS module — because small modules can have a surprisingly high performance cost101 depending on your choice of bundler and module […]

Reply

Posted by Front-End Performance Checklist 2017 (PDF, Apple Pages) - on December 21, 2016 at 2:41 AM

[…] converts ECMAScript 2015 modules into one big CommonJS module — because small modules can have a surprisingly high performance cost101 depending on your choice of bundler and module […]

Reply

Posted by Front-End Performance Checklist 2017 (PDF, Apple Pages) - Webdesign Journal on December 21, 2016 at 3:06 AM

[…] converts ECMAScript 2015 modules into one big CommonJS module — because small modules can have a surprisingly high performance cost103 depending on your choice of bundler and module […]

Reply

Posted by Front-End Performance Checklist 2017 (PDF, Apple Pages) | Tech-Chat on December 21, 2016 at 8:34 AM

[…] converts ECMAScript 2015 modules into one big CommonJS module — because small modules can have a surprisingly high performance cost103 depending on your choice of bundler and module […]

Reply

Posted by Front-End Performance Checklist 2017 (PDF, Apple Pages) on December 21, 2016 at 10:04 AM

[…] converts ECMAScript 2015 modules into one big CommonJS module — because small modules can have a surprisingly high performance cost103 depending on your choice of bundler and module […]

Reply

Posted by Front-End Performance Checklist 2017 – Smashing Magazine – VaughnPaul.com on January 6, 2017 at 11:44 AM

[…] converts ECMAScript 2015 modules into one big CommonJS module — because small modules can have a surprisingly high performance cost103 depending on your choice of bundler and module […]

Reply

Posted by Front-End Performance Checklist 2017 (PDF, Apple Pages) – Web Guy Help on January 10, 2017 at 9:33 AM

[…] converts ECMAScript 2015 modules into one big CommonJS module — because small modules can have a surprisingly high performance cost103 depending on your choice of bundler and module […]

Reply

Posted by 2017 ▯▯▯▯▯▯▯▯ - ▯▯▯▯ on January 15, 2017 at 10:53 PM

[…] ▯▯▯▯▯ […]

Reply

Posted by #6 ניוזלטר – חדשות מעולם הפרונט אנד | Outbrain Techblog on March 21, 2017 at 2:01 AM

[…] מאמר מעניין על ארכיטקטורה של מודולים קטנים ומה העלות של שימוש בהם **https://nolanlawson.com/2016/08/15/the-cost-of-small-modules/** […]

Reply

Posted by Angular 4 réduit la taille du code généré, et ajoute des fonctionnalités on March 27, 2017 at 1:55 AM

[…] significatif lors de l’exécution d’un template. On va également retrouver des versions « Flat » des modules Angular, ou encore disposer d’une plus grande souplesse lors de l’utilisation […]

Reply

Posted by Angular4.0.0▯▯▯▯▯▯▯▯▯▯▯▯ | ▯▯▯▯▯Cadenza▯▯▯▯▯▯▯ on April 9, 2017 at 6:20 AM

[…] Flat ES Modules▯▯▯▯▯▯▯▯▯ The cost of small modules▯▯▯▯▯▯▯▯▯▯▯▯ […]

Reply

Posted by Angular 4 – estude isso on April 13, 2017 at 12:57 PM

[…] Leia mais sobre a importância dos módulos Flat ES “The cost of small modules”. […]

Reply

Posted by webpack▯Rollup▯▯▯▯▯▯▯▯▯ | ▯▯▯▯▯▯▯ | POSTD on June 7, 2017 at 2:31 AM

[…] ▯▯▯Rollup▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯ES2015▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯JavaScript▯▯▯▯▯▯▯▯▯▯▯▯▯▯ ▯▯▯▯▯▯webpack▯▯require▯▯▯▯▯▯▯▯▯▯1 ▯1▯▯ […]

Reply

Posted by Webpack ▯ Rollup ▯▯▯▯▯▯▯ – WEB▯▯▯▯ - ▯▯▯▯▯▯▯▯▯▯▯▯▯▯ on July 24, 2017 at 9:18 PM

[…] Rollup ▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯ ES2015 ▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯ JavaScript ▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯ – ▯▯ webpack – ▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯ require ▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯webpack ▯▯▯▯▯▯▯▯▯▯ ▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯ […]

Reply

Posted by Webpack ▯ Rollup ▯▯▯▯▯▯▯ – ▯▯▯▯ on July 24, 2017 at 11:42 PM

[…] ▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯ ▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯ […]

Reply

Posted by Creative techniques for writing modular code – My WordPress Website on October 18, 2017 at 9:02 AM

[…] There are tons of JS modules out there that include just a single small function. Each module requires additional parsing and processing time, and includes their own header in the code, so using a lot of small modules will add overhead to your build system and increase your bundle si… […]

Reply

Posted by Front End Performance Checklist 2017 [PDF, Apple Pages] | Design Guru on December 4, 2017 at 3:00 PM

[…] converts ECMAScript 2015 modules into one big CommonJS module — because small modules can have a surprisingly high performance cost depending on your choice of bundler and module […]

Reply

Posted by □□□□□□□ | <meta>□□□ on December 23, 2017 at 7:35 PM

[…] The cost of small modules […]

Reply

Posted by Front-End Performance Checklist 2018 [PDF, Apple Pages] | Domain Industry News on January 3, 2018 at 7:49 AM

[…] ECMAScript 2015 modules into one big CommonJS module — because small modules can have a surprisingly high performance cost depending on your choice of bundler and module […]

Reply

Posted by Front-End Performance Checklist 2018 [PDF, Apple Pages] | Tech News on January 3, 2018 at 7:53 AM

[…] converts ECMAScript 2015 modules into one big CommonJS module — because small modules can have a surprisingly high performance cost depending on your choice of bundler and module […]

Reply

Posted by January 3, 2018 Front-End Performance Checklist 2018 [PDF, Apple Pages] — Smashing Magazine on January 3, 2018 at 8:34 AM

[…] converts ECMAScript 2015 modules into one big CommonJS module — because small modules can have a surprisingly high performance cost depending on your choice of bundler and module […]

Reply

Posted by Front-End Performance Checklist - Scripts Mix on January 4, 2018 at 12:26 PM

[…] ECMAScript 2015 modules into one big CommonJS module — because small modules can have a surprisingly high performance cost depending on your choice of bundler and module […]

Reply

Posted by Front-End Performance Checklist 2018 [PDF, Apple Pages] — Smashing Magazine | OntheWebIT.com - Website Development and Design on January 4, 2018 at 12:48 PM

[…] converts ECMAScript 2015 modules into one big CommonJS module — because small modules can have a surprisingly high performance cost depending on your choice of bundler and module […]

Reply

Posted by What is the difference between Bower and npm? - QuestionFocus on February 8, 2018 at 9:31 PM

[…] that Webpack and rollup are widely regarded to be better than Browserify as of Aug […]

Reply

Posted by 2017□□□□□□□□ | □□□□ on June 3, 2018 at 5:40 AM

[…] Rollup□□□□□□□□Browserify□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□Rollupify□□□ECMAScript2015□□□ □□□□□□□CommonJS□□——□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ […]

Reply

Leave a Reply

Enter your comment here...