Speeding up the JavaScript ecosystem - one library at a time

written by **@marvinhagemeist** 29 November 2022

📖 tl;dr: Most popular libraries can be sped up by avoiding unnecessary type conversions or by avoiding creating functions inside functions.

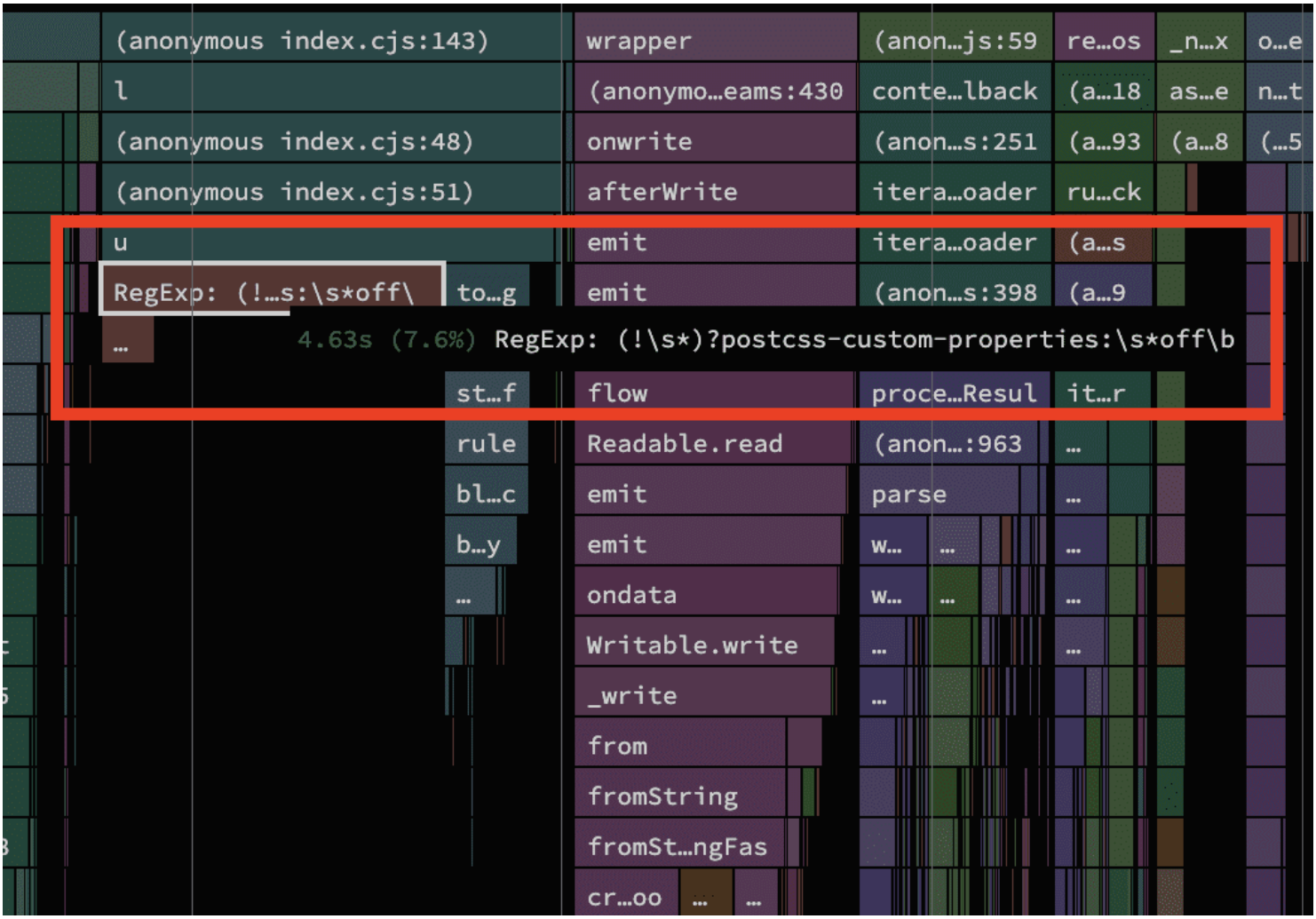
Whilst the trend is seemingly to rewrite every JavaScript build tool in other languages such as Rust or Go, the current JavaScript-based tools could be a lot faster. The build pipeline in a typical frontend project is usually composed of many different tools working together. But the diversification of tools makes it a little harder to spot performance problems for tooling maintainers as they need to know which tools their own ones are frequently used with.

Although JavaScript is certainly slower than Rust or Go from a pure language point of view, the current JavaScript tools could be improved considerably. Sure JavaScript is slower, but it shouldn't be *that* slow in comparison as it is today. JIT engines are crazy fast these days!

Curiosity lead me down the path of spending some time profiling common JavaScript-based tools to kinda see where all that time was spent. Let's start with PostCSS, a very popular parser and transpiler for anything CSS.

Saving 4.6s in PostCSS

There is a very useful plugin called postcss-custom-properties that adds basic support for CSS Custom Properties in older browsers. Somehow it showed up very prominently in traces with a costly 4.6s being attributed to a single regex that it uses internally. That looked odd.



The regex looks suspiciously like something that searches for a particular comment value to change the plugin's behaviour, similar to the ones from eslint used to disable specific linting rules. It's not mentioned in their README, but a peek into the source code confirmed that assumption.

The place where the regex is created is part of a function that checks if a CSS rule or declaration is preceeded by said comment.

```js
function isBlockIgnored(ruleOrDeclaration) {
  const rule = ruleOrDeclaration.selector
    ? ruleOrDeclaration
    : ruleOrDeclaration.parent;
```

```
  return /(!\s*)?postcss-custom-properties:\s*off\b/i.test(rule.toString());
}
```

The `rule.toString()` call caught my eye pretty quickly. Places where one type is cast to another are usually worth another look if you're tackling performance as not having to do the conversion always saves time. What was interesting in this scenario is that the `rule` variable always holds an `object` with a custom `toString` method. It was never a string to begin with, so we know that we're always paying a bit of serialization cost here to be able to test the regex against. From experience I knew that matching a regex against many short strings is a lot slower than matching it against few long ones. This is a prime candidate waiting to be optimized!

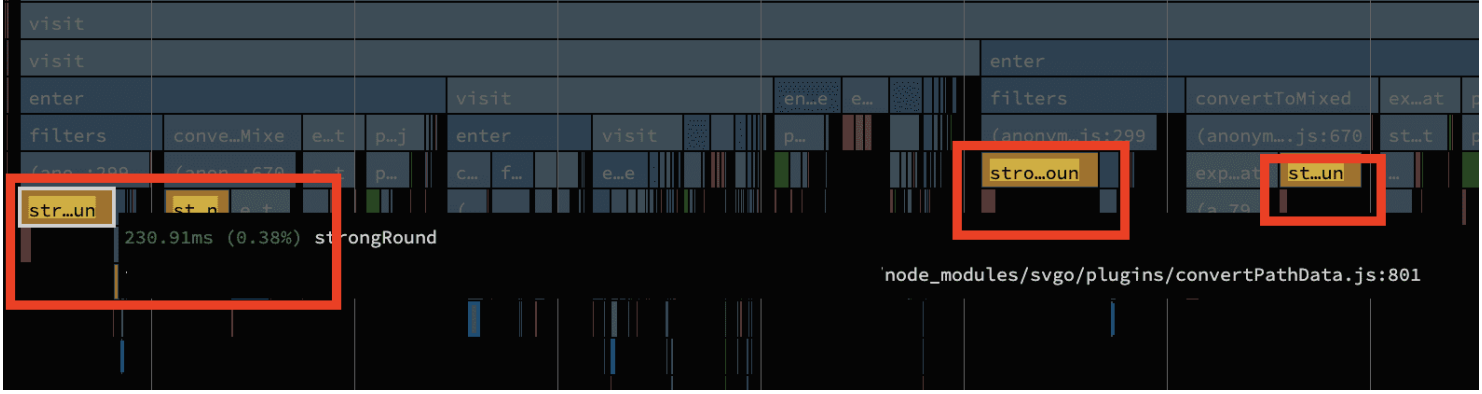
The rather troubling aspect of this code is that every input file has to pay this cost, regardless of whether it has a postcss comment or not. Knowing that running one regex over a long string is cheaper than the repeated regex over short strings and the serialization cost, we can guard this function to avoid even having to call `isBlockIgnored` if we know that the file doesn't contain any postcss comments.

With **the fix** applied, the build time went down by a whopping **4.6s**!

Optimizing SVG compression speed

Next up is **SVGO**, a library for compressing SVG files. It's pretty awesome and a staple for projects with lots of SVG icons. The CPU profile reveiled that **3.1s** was spent in compressing SVGs though. Can we speed that up?

Searching a bit through the profiling data there was one function that stood out: `strongRound`. What's more is that function was always followed by a little bit of GC cleanup shortly after (see the small red boxes).



Consider my curiosity piqued! Let's pull up the **source on GitHub**:

```js
/**
 * Decrease accuracy of floating-point numbers
 * in path data keeping a specified number of decimals.
 * Smart rounds values like 2.3491 to 2.35 instead of 2.349.
 */
function strongRound(data: number[]) {
  for (var i = data.length; i-- > 0; ) {
    if (data[i].toFixed(precision) != data[i]) {
      var rounded = +data[i].toFixed(precision - 1);
      data[i] =
        +Math.abs(rounded - data[i]).toFixed(precision + 1) >= error
          ? +data[i].toFixed(precision)
          : rounded;
    }
  }
  return data;
}
```

Aha, so it's a function that is used to compress numbers, which there are a lot of in any typical SVG file. The function receives an array of `numbers` and is expected to mutate its entries. Let's take a look at the type of variables that are used in its implementation. With a bit of closer inspection we notice that there is a lot of back and forth casting between string and numbers.

```js
function strongRound(data: number[]) {
  for (var i = data.length; i-- > 0; ) {
    // Comparison between string and number -> string is cast to number
    if (data[i].toFixed(precision) != data[i]) {
```

```
      // Creating a string from a number that's casted immediately
      // back to a number
      var rounded = +data[i].toFixed(precision - 1);
      data[i] =
        // Another number that is casted to a string and directly back
        // to a number again
        +Math.abs(rounded - data[i]).toFixed(precision + 1) >= error
          ? // This is the same value as in the if-condition before,
            // just casted to a number again
            +data[i].toFixed(precision)
          : rounded;
    }
  }
  return data;
}
```

The act of rounding numbers seems like something that can be done with a little bit of math alone, without having to convert numbers to strings. As a general rule of thumb a good chunk of optimizations are about expressing things in numbers, the main reason being that CPUs are crazy good at working with numbers. With a few changes here and there we can ensure that we always stay in number-land and thus avoid the string casting entirely.

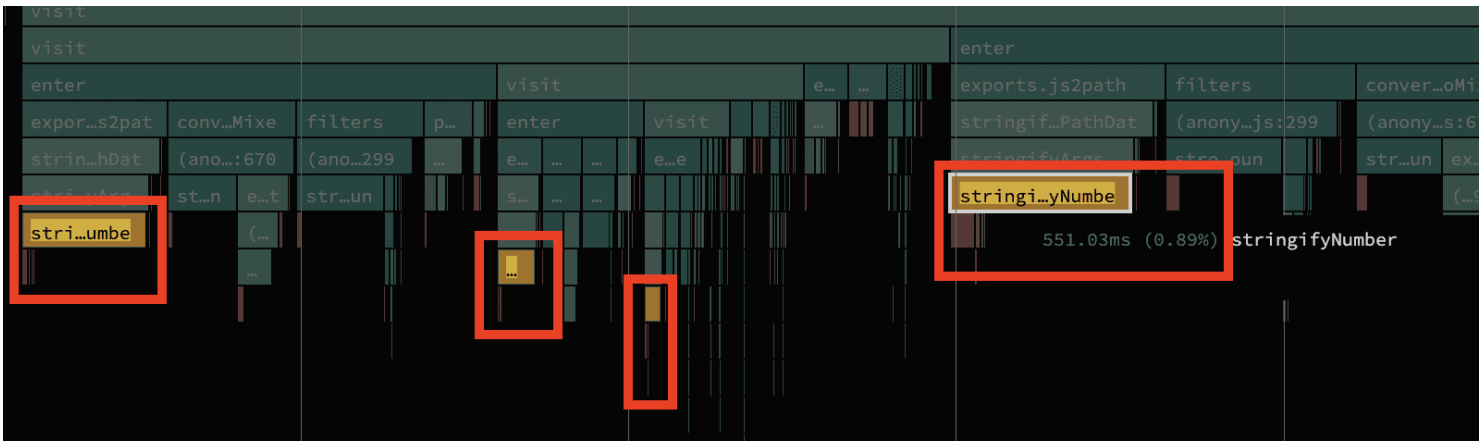
```js
// Does the same as `Number.prototype.toFixed` but without casting
// the return value to a string.
function toFixed(num, precision) {
  const pow = 10 ** precision;
  return Math.round(num * pow) / pow;
}

// Rewritten to get rid of all the string casting and call our own
// toFixed() function instead.
function strongRound(data: number[]) {
  for (let i = data.length; i-- > 0; ) {
    const fixed = toFixed(data[i], precision);
    // Look ma, we can now use a strict equality comparison!
    if (fixed !== data[i]) {
      const rounded = toFixed(data[i], precision - 1);
      data[i] =
        toFixed(Math.abs(rounded - data[i]), precision + 1) >= error
          ? fixed // We can now reuse the earlier value here
          : rounded;
    }
```

```
  }
  return data;
}
```

Running the profiling again confirmed that we were able to speed up build times by about **1.4s**! I've filed a PR upstream for that too.

Regexes on short strings (part 2)

In close vicinity to `strongRound` another function looked suspicious as it takes up nearly a whole second (0.9s) to complete.



Similar to `stringRound` this function compresses numbers too, but with the added trick that we can drop the leading zero if the number has decimals and is smaller than 1 and bigger than -1. So `0.5` can be compressed to `.5` and `-0.2` to `-.2` respectively. In particular the last line looks of interest.

```js
const stringifyNumber = (number: number, precision: number) => {
  // ...snip

  // remove zero whole from decimal number
  return number.toString().replace(/^0\./, ".").replace(/^-0\./, "-.");
};
```

Here we are converting a number to a string and calling a regex on it. It's extremely likely that the string version of the number will be a short string. And we know that a number can not be both `n > 0 && n < 1` and `n > -1 && < 0` at the same time. Not even `NaN` has that power! From that we can deduce that either only one of the regexes matches or none of them, but never both. At least one of the `.replace` calls is always wasted.

We can optimize that by differentiating between those cases by hand. Only if we know that we're dealing with a number that has a leading `0` should we apply our replacement logic. Those number checks are quicker than doing a regex search.

```js
const stringifyNumber = (number: number, precision: number) => {
  // ...snip

  // remove zero whole from decimal number
  const strNum = number.toString();
  // Use simple number checks
  if (0 < num && num < 1) {
    return strNum.replace(/^0\./, ".");
  } else if (-1 < num && num < 0) {
    return strNum.replace(/^-0\./, "-.");
  }
  return strNum;
};
```

We can go one step further and get rid of the regex searches entirely as we know with 100% certainty where the leading `0` is in the string and thus can manipulate the string directly.

```js
const stringifyNumber = (number: number, precision: number) => {
  // ...snip

  // remove zero whole from decimal number
  const strNum = number.toString();
  if (0 < num && num < 1) {
    // Plain string processing is all we need
    return strNum.slice(1);
  } else if (-1 < num && num < 0) {
    // Plain string processing is all we need
```

```
    return "-" + strNum.slice(2);
  }
  return strNum;
};
```

Since there is a separate function to trim the leading `0` in svgo's codebase already, we can leverage that instead. Another 0.9s saved! **Upstream PR**.

Inline functions, inline caches and recursion

One function called `monkeys` intrigued me due to its name alone. In traces I could see that it was called multiple times inside itself, which is a strong indicator that some sort of recursion is happening here. It's often used to traverse a tree-like structure. Whenever some sort of traversal is used, there is a likelihood that it's somewhat in the "hot" path of the code. That's not true for every scenario but in my experience it has been a good rule of thumb.

```js
function perItem(data, info, plugin, params, reverse) {
  function monkeys(items) {
    items.children = items.children.filter(function (item) {
      // reverse pass
      if (reverse && item.children) {
        monkeys(item);
      }
      // main filter
      let kept = true;
      if (plugin.active) {
        kept = plugin.fn(item, params, info) !== false;
      }
      // direct pass
      if (!reverse && item.children) {
        monkeys(item);
      }
      return kept;
    });
    return items;
  }
```

```
  return monkeys(data);
}
```

Here we have a function that creates another function inside its body which recalls the inner function again. If I had to guess, I'd assume that this was done here to save some keystrokes by not having to pass around all arguments again. Thing is that functions that are created inside other functions are pretty difficult to optimize when the outer function is called frequently.

```js
function perItem(items, info, plugin, params, reverse) {
  items.children = items.children.filter(function (item) {
    // reverse pass
    if (reverse && item.children) {
      perItem(item, info, plugin, params, reverse);
    }
    // main filter
    let kept = true;
    if (plugin.active) {
      kept = plugin.fn(item, params, info) !== false;
    }
    // direct pass
    if (!reverse && item.children) {
      perItem(item, info, plugin, params, reverse);
    }
    return kept;
  });
  return items;
}
```

We can get rid of the inner function by always passing all arguments explicitly vs capturing them by closure like before. The impact of this change is rather minor, but in total it saved another **0.8s**.

Luckily this is already addressed in the new major `3.0.0` release, but it will take a bit until the ecosystem switches to the new version.

Beware of for...of transpilation

A nearly identical problem occurs in `@vanilla-extract/css`. The published package ships with the following piece of code:

```js
class ConditionalRuleset {
  getSortedRuleset() {
    //...
    var _loop = function _loop(query, dependents) {
      doSomething();
    };

    for (var [query, dependents] of this.precedenceLookup.entries()) {
      _loop(query, dependents);
    }
    //...
  }
}
```

What's interesting about this function is that it's not present in the original source code. In the original source it's a standard `for...of` loop.

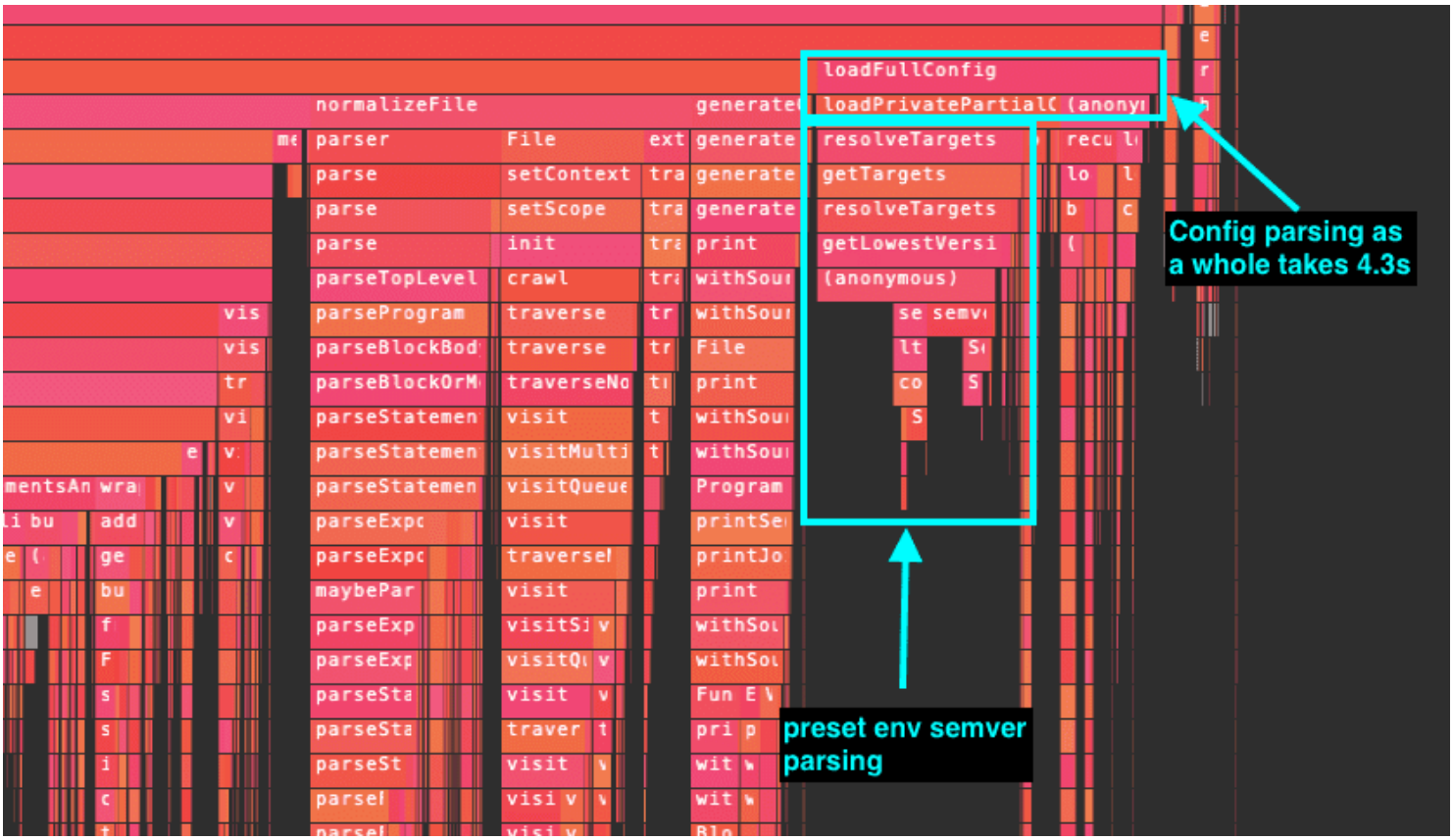
```js
class ConditionalRuleset {
  getSortedRuleset() {
    //...
    for (var [query, dependents] of this.precedenceLookup.entries()) {
      doSomething();
    }
    //...
  }
}
```

I couldn't replicate this issue in babel or typescript's repl, but I can confirm that it's introduced by their build pipeline. Given that it seems to be a shared abstraction over their build tool I'd assume that a few more projects are affected by this. So for now I just

patched the package locally inside `node_modules` and was happy to see that this improved the build times by another `0.9s`.

The curious case of semver

With this one I'm not sure if I've configured something wrong. Essentially, the profile showed that the whole babel configuration was always read anew whenever it transpiled a file.



It's a bit hard to see in the screenshot, but one of the functions taking up much time was code from the `semver` package, the same package that's used in npm's cli. Huh? What has semver to do with babel? It took a while until it dawned on me: It's for parsing the browserlist target for `@babel/preset-env`. Although the browserlist settings might look pretty short, ultimately they were expanded to about 290 individual targets.

That alone isn't enough for concern, but it's easy to miss the allocation cost when using validation functions. It's a bit spread out in babel's code base, but essentially the versions of the browser targets are converted to semver strings `"10" -> "10.0.0"` and then validated. Some of those version numbers already match the semver format. These

versions and sometimes version ranges are compared against each other until we find the lowest common feature set we need to transpile for. There is nothing wrong with this approach.

Performance problems arise here, because the semver versions are stored as a `string` instead of the parsed semver data type. This means that every call to `semver.valid('1.2.3')` will create a new semver instance and immediately destroy it. Same is true when comparing semver versions when using strings: `semver.lt('1.2.3', '9.8.7')`. And *that* is why we're seeing semver so prominently in the traces.

By patching that locally in `node_modules` again, I was able to reduce the build time by another `4.7s`.

Conclusion

At this point I stopped looking, but I'd assume that you'll find more of these minor performance issues in popular libraries. Today we mainly looked at some build tools, but UI components or other libraries usually have the same low hanging performance issues.

Will this be enough to match Go's or Rust's performance? Unlikely, but the thing is that the current JavaScript tools could be faster than they are today. And the things we looked at in this post are more or less just the tip of the iceberg.