Blazingly fast parsing, part 2: lazy parsing

Published 15 April 2019 · Tagged with internals parsing

This is the second part of our series explaining how V8 parses JavaScript as fast as possible. The first part explained how we made V8's scanner fast.

Parsing is the step where source code is turned into an intermediate representation to be consumed by a compiler (in V8, the bytecode compiler Ignition). Parsing and compiling happens on the critical path of web page startup, and not all functions shipped to the browser are immediately needed during startup. Even though developers can delay such code with async and deferred scripts, that's not always feasible. Additionally, many web pages ship code that's only used by certain features which may not be accessed by a user at all during any individual run of the page.

Eagerly compiling code unnecessarily has real resource costs:

- CPU cycles are used to create the code, delaying the availability of code that's actually needed for startup.
- Code objects take up memory, at least until bytecode flushing decides that the code isn't currently needed and allows it to be garbage-collected.
- Code compiled by the time the top-level script finishes executing ends up being cached on disk, taking up disk space.

For these reasons, all major browsers implement *lazy parsing*. Instead of generating an abstract syntax tree (AST) for each function and then compiling it to bytecode, the parser can decide to "pre-parse" functions it encounters instead of fully parsing them. It does so by switching to the preparser, a copy of the parser that does the bare minimum needed to be able to otherwise skip over the function. The preparser verifies that the functions it skips are syntactically valid, and produces all the information needed for the outer functions to be compiled correctly. When a preparsed function is later called, it is fully parsed and compiled on-demand.

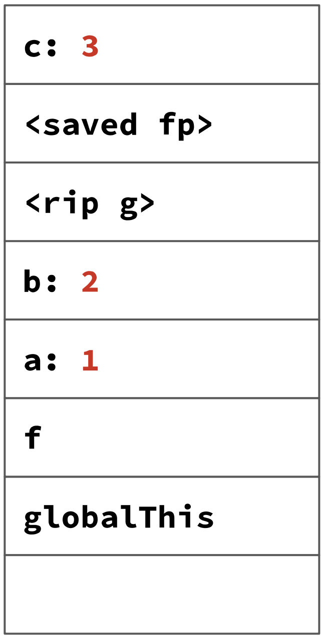
Variable allocation

The main thing that complicates pre-parsing is variable allocation.

For performance reasons, function activations are managed on the machine stack. E.g., if a function `g` calls a function `f` with arguments `1` and `2`:

```
function f(a, b) {
  const c = a + b;
  return c;
}

function g() {
  return f(1, 2);
  // The return instruction pointer of `f` now points here
  // (because when `f` `return`s, it returns here).
}
```

First the receiver (i.e. the `this` value for `f`, which is `globalThis` since it's a sloppy function call) is pushed on the stack, followed by the called function `f`. Then arguments `1` and `2` are pushed on the stack. At that point the function `f` is called. To execute the call, we first save the state of `g` on the stack: the "return instruction pointer" (`rip`; what code we need to return to) of `f` as well as the "frame pointer" (`fp`; what the stack should look like on return). Then we enter `f`, which allocates space for the local variable `c`, as well as any temporary space it may need. This ensures that any data used by the function disappears when the function activation goes out of scope: it's simply popped from the stack.



c: 3

<saved fp>

<rip g>

b: 2

a: 1

f

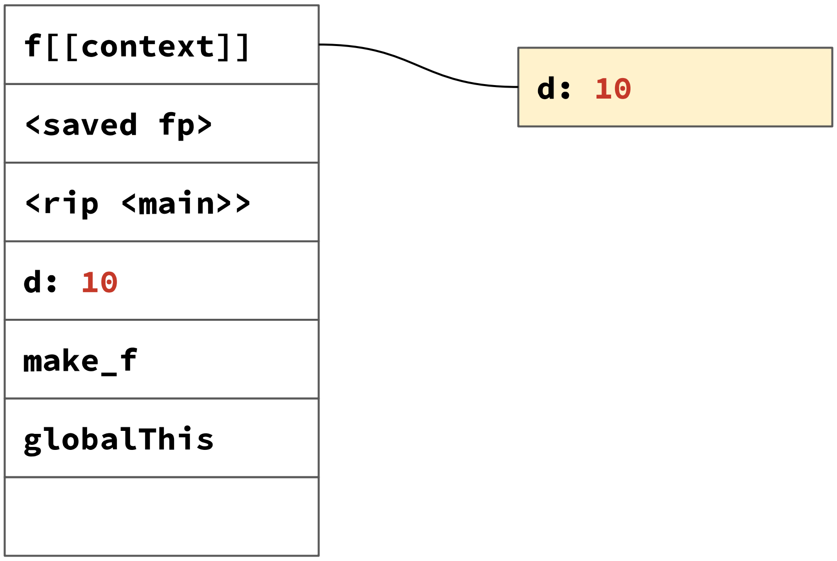
globalThis

Stack layout of a call to function `f` with arguments `a`, `b`, and local variable `c` allocated on the stack.

The problem with this setup is that functions can reference variables declared in outer functions. Inner functions can outlive the activation in which they were created:

```
function make_f(d) { // ← declaration of `d`
  return function inner(a, b) {
    const c = a + b + d; // ← reference to `d`
    return c;
  };
}

const f = make_f(10);

function g() {
  return f(1, 2);
}
```

In the above example, the reference from `inner` to the local variable `d` declared in `make_f` is evaluated after `make_f` has returned. To implement this, VMs for languages with lexical closures allocate variables referenced from inner functions on the heap, in a structure called a “context”.



Stack layout of a call to `make_f` with the argument copied to a context allocated on the heap for later use by `inner` that captures `d`.

This means that for each variable declared in a function, we need to know whether an inner function references the variable, so we can decide whether to allocate the variable on the stack or in a heap-allocated context. When we evaluate a function literal, we allocate a closure that points both to the code for the function, as well as the current context: the object that contains the variable values it may need access to.

Long story short, we do need to track at least variable references in the preparser.

If we'd only track references though, we would overestimate what variables are referenced. A variable declared in an outer function could be shadowed by a redeclaration in an inner function, making a reference from that inner function target the inner declaration, not the outer declaration. If we'd unconditionally allocate the outer variable in the context, performance would suffer. Hence for variable allocation to properly work with preparsing, we need to make sure that preparsed functions properly keep track of variable references as well as declarations.

Top-level code is an exception to this rule. The top-level of a script is always heap-allocated, since variables are visible across scripts. An easy way to get close to a well-working architecture is to simply run the preparser without variable tracking to fast-parse top-level functions; and to use the full parser for inner functions, but skip compiling them. This is more costly than preparsing since we unnecessarily build up an entire AST, but it gets us up and running. This is exactly what V8 did up to V8 v6.3 / Chrome 63.

Teaching the preparser about variables

Tracking variable declarations and references in the preparser is complicated because in JavaScript it isn't always clear from the start what the meaning of a partial expression is. E.g., suppose we have a function `f` with a parameter `d`, which has an inner function `g` with an expression that looks like it might reference `d`.

```
function f(d) {
  function g() {
    const a = ({ d }
```

It could indeed end up referencing `d`, because the tokens we saw are part of a destructuring assignment expression.

```
function f(d) {
  function g() {
    const a = ({ d } = { d: 42 });
    return a;
  }
  return g;
}
```

It could also end up being an arrow function with a destructuring parameter `d`, in which case the `d` in `f` isn't referenced by `g`.

```
function f(d) {
  function g() {
    const a = ({ d }) => d;
    return a;
  }
  return [d, g];
}
```

Initially our preparser was implemented as a standalone copy of the parser without too much sharing, which caused the two parsers to diverge over time. By rewriting the parser and preparser to be based on a `ParserBase` implementing the curiously recurring template pattern, we managed to maximize sharing while keeping the performance benefits of separate copies. This greatly simplified adding full variable tracking to the preparser, since a large part of the implementation can be shared between the parser and the preparser.

Actually it was incorrect to ignore variable declarations and references even for top-level functions. The ECMAScript spec requires various types of variable conflicts to be detected upon first parse of the script. E.g., if a variable is twice declared as a lexical variable in the same scope, that is considered an early `SyntaxError`. Since our preparser simply skipped over variable declarations, it would incorrectly allow the code during preparse. At the time we considered that the performance win warranted the spec violation. Now that the preparser tracks variables properly, however, we eradicated this entire class of variable resolution-related spec violations at no significant performance cost.

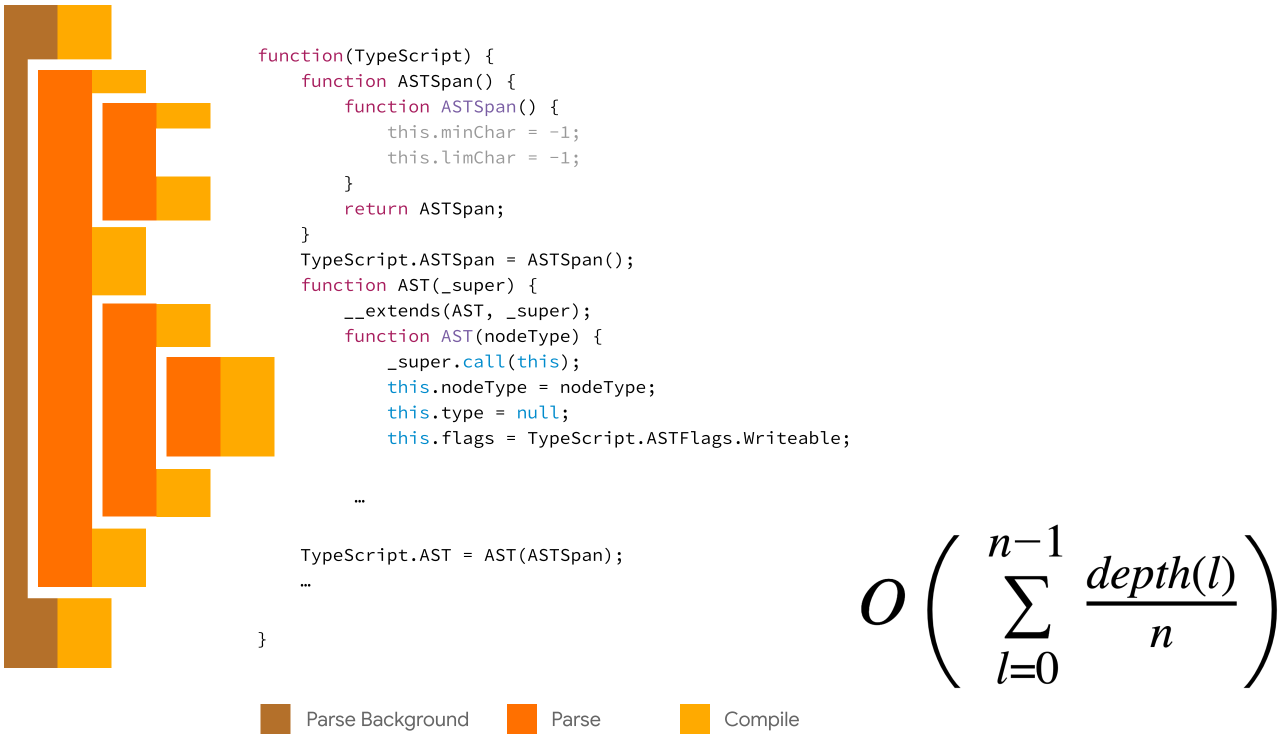
Skipping inner functions

As mentioned earlier, when a preparsed function is called for the first time, we parse it fully and compile the resulting AST to bytecode.

```
// This is the top-level scope.
function outer() {
  // preparsed
  function inner() {
    // preparsed
  }
}

outer(); // Fully parses and compiles `outer`, but not `inner`.
```

The function directly points to the outer context which contains the values of variable declarations that need to be available to inner functions. To allow lazy compilation of functions (and to support the debugger), the context points to a metadata object called `ScopeInfo`. `ScopeInfo` objects describe what variables are listed in a context. This means that while compiling inner functions, we can compute where variables live in the context chain.

To compute whether or not the lazy compiled function itself needs a context, though, we need to perform scope resolution again: We need to know whether functions nested in the lazy-compiled function reference the variables declared by the lazy function. We can figure this out by re-preparsing those functions. This is exactly what V8 did up to V8 v6.3 / Chrome 63. This is not ideal performance-wise though, as it makes the relation between source size and parse cost nonlinear: we would preparse functions as many times as they are nested. In addition to natural nesting of dynamic programs, JavaScript packers commonly wrap code in “immediately-invoked function expressions” (IIFEs), making most JavaScript programs have multiple nesting layers.



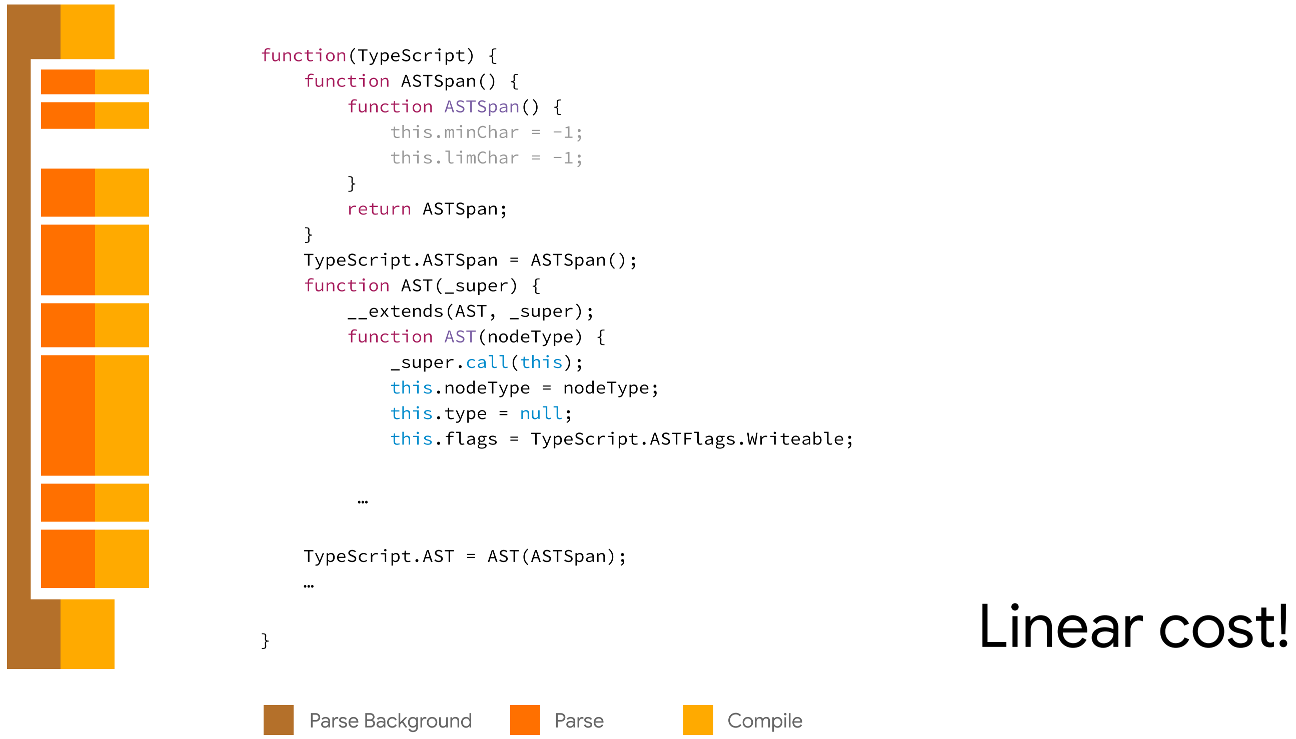
```
function(TypeScript) {
    function ASTSpan() {
        function ASTSpan() {
            this.minChar = -1;
            this.limChar = -1;
        }
        return ASTSpan;
    }
    TypeScript.ASTSpan = ASTSpan();
    function AST(_super) {
        __extends(AST, _super);
        function AST(nodeType) {
            _super.call(this);
            this.nodeType = nodeType;
            this.type = null;
            this.flags = TypeScript.ASTFlags.Writeable;


        …

    TypeScript.AST = AST(ASTSpan);
    …

}
```

$$O\left(\sum_{l=0}^{n-1} \frac{depth(l)}{n}\right)$$

Each reparse adds at least the cost of parsing the function.

To avoid the nonlinear performance overhead, we perform full scope resolution even during preparsing. We store enough metadata so we can later simply *skip* inner functions, rather than having to re-preparse them. One way would be to store variable names referenced by inner functions. This is expensive to store and requires us to still duplicate work: we have already performed variable resolution during preparse.

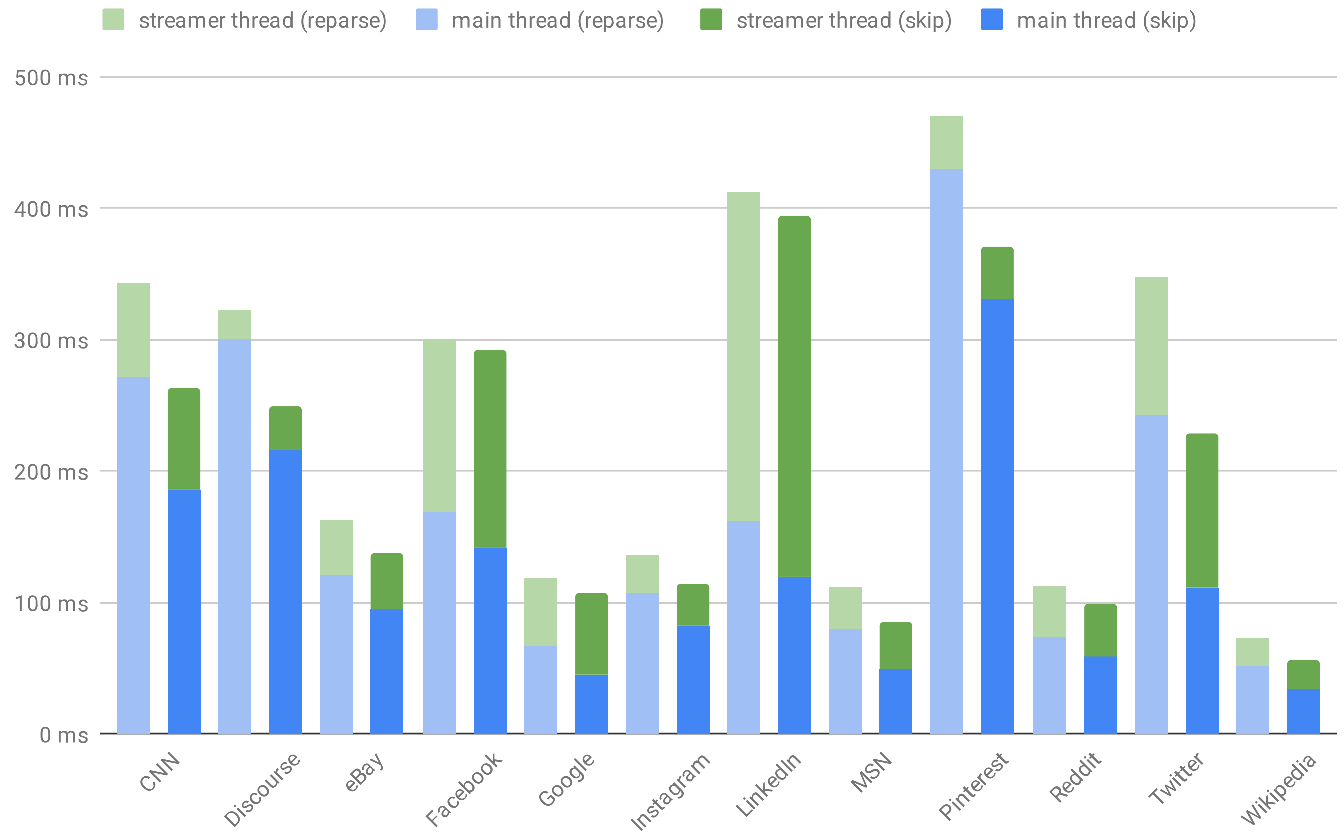
Instead, we serialize where variables are allocated as a dense array of flags per variable. When we lazy-parse a function, variables are recreated in the same order as the preparser saw them, and we can simply apply the metadata to the variables. Now that the function is compiled, the variable allocation metadata is not needed anymore and can be garbage-collected. Since we only need this metadata for functions that actually contain inner functions, a large fraction of all functions does not even need this metadata, significantly reducing the memory overhead.



```
function(TypeScript) {
    function ASTSpan() {
        function ASTSpan() {
            this.minChar = -1;
            this.limChar = -1;
        }
        return ASTSpan;
    }
    TypeScript.ASTSpan = ASTSpan();
    function AST(_super) {
        __extends(AST, _super);
        function AST(nodeType) {
            _super.call(this);
            this.nodeType = nodeType;
            this.type = null;
            this.flags = TypeScript.ASTFlags.Writeable;


         …


    TypeScript.AST = AST(ASTSpan);
    …

}
```

By keeping track of metadata for preparsed functions we can completely skip inner functions.

The performance impact of skipping inner functions is, just like the overhead of re-preparsing inner functions, nonlinear. There are sites that hoist all their functions to the top-level scope. Since their nesting level is always 0, the overhead is always 0. Many modern sites, however, do actually deeply nest functions. On those sites we saw significant improvements when this feature launched in V8 v6.3 / Chrome 63. The main advantage is that now it doesn't matter anymore how deeply nested the code is: any function is at most preparsed once, and fully parsed once[1].



Main thread and off-the-main-thread parse time, before and after launching the “skipping inner functions” optimization.

Possibly-Invoked Function Expressions

As mentioned earlier, packers often combine multiple modules in a single file by wrapping module code in a closure that they immediately call. This provides isolation for the modules, allowing them to run as if they are the only code in the script. These functions are essentially nested scripts; the functions are immediately called upon script execution. Packers commonly ship *immediately-invoked function expressions* (IIFEs; pronounced “iffies”) as parenthesized functions: `(function(){…})()`.

Since these functions are immediately needed during script execution, it’s not ideal to preparse such functions. During top-level execution of the script we immediately need the function to be compiled, and we fully parse and compile the function. This means that the faster parse we did earlier to try to speed up startup is guaranteed to be an unnecessary additional cost to startup.

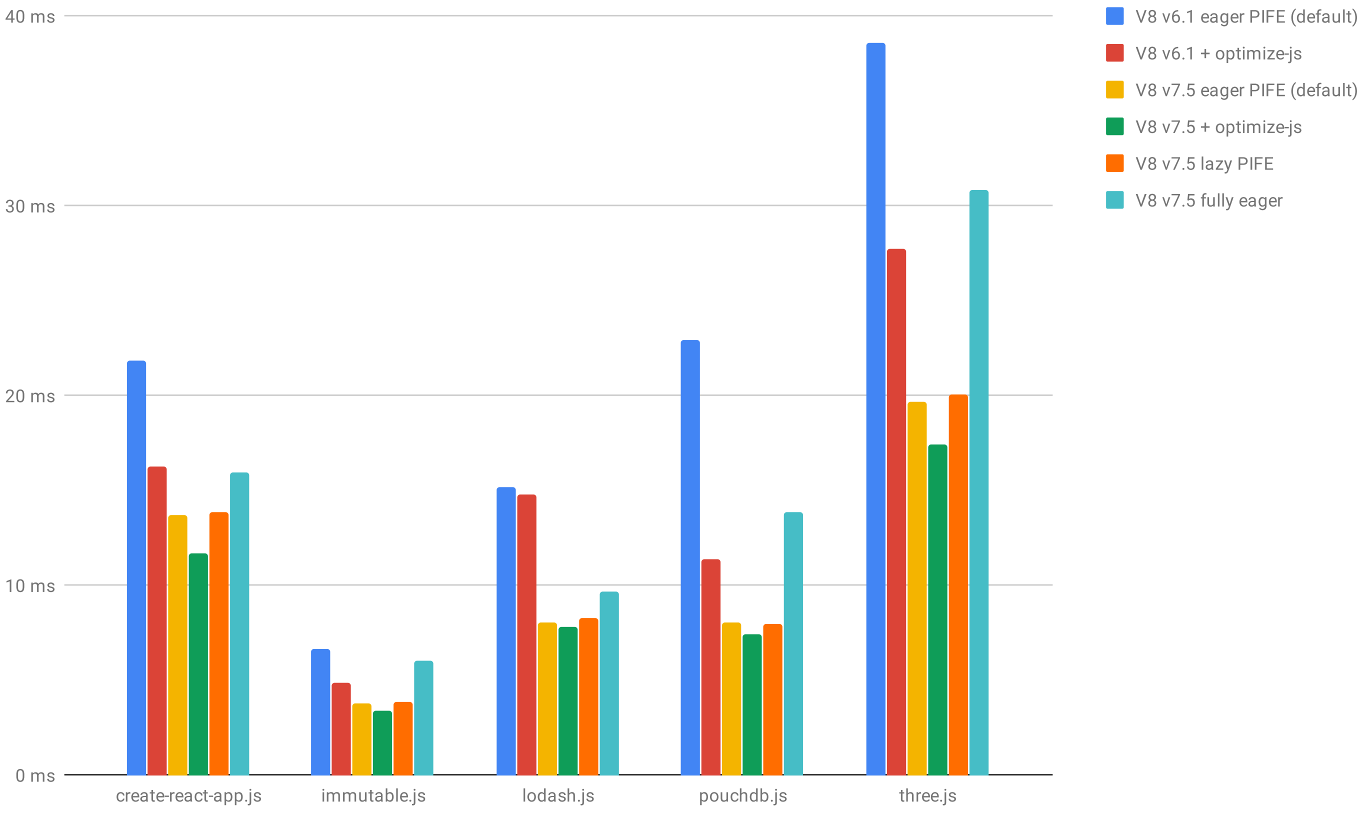
Why don’t you simply compile called functions, you might ask? While it’s typically straight-forward for a developer to notice when a function is called, this is not the case for the parser. The parser needs to decide — before it even starts parsing a function! — whether it wants to eagerly compile the function or defer compilation. Ambiguities in the syntax make it difficult to simply fast-scan to the end of the function, and the cost quickly resembles the cost of regular preparsing.

For this reason V8 has two simple patterns it recognizes as *possibly-invoked function expressions* (PIFEs; pronounced “piffies”), upon which it eagerly parses and compiles a function:

- If a function is a parenthesized function expression, i.e. `(function(){…})`, we assume it will be called. We make this assumption as soon as we see the start of this pattern, i.e. `(function`.
- Since V8 v5.7 / Chrome 57 we also detect the pattern `!function(){…}(),function(){…}(),function(){…}()` generated by UglifyJS. This detection kicks in as soon as we see `!function`, or `,function` if it immediately follows a PIFE.

Since V8 eagerly compiles PIFEs, they can be used as profile-directed feedback [2], informing the browser which functions are needed for startup.

At a time when V8 still reparsed inner functions, some developers had noticed the impact of JS parsing on startup was pretty high. The package `optimize-js` turns functions into PIFEs based on static heuristics. At the time the package was created, this had a huge impact on load performance on V8. We’ve replicated these results by running the benchmarks provided by `optimize-js` on V8 v6.1, only looking at minified scripts.

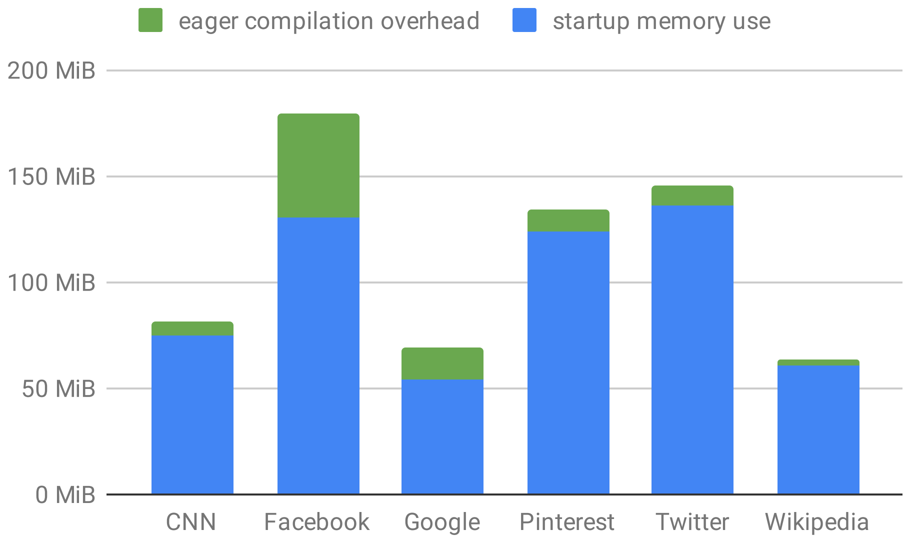


Eagerly parsing and compiling PIFEs results in slightly faster cold and warm startup (first and second page load, measuring total parse + compile + execute times). The benefit is much smaller on V8 v7.5 than it used to be on V8 v6.1 though, due to significant improvements to the parser.

Nevertheless, now that we don't reparse inner functions anymore and since the parser has gotten much faster, the performance improvement obtained through `optimize-js` is much reduced. The default configuration for v7.5 is in fact already much faster than the optimized version running on v6.1 was. Even on v7.5 it can still makes sense to use PIFEs sparingly for code that is needed during startup: we avoid preparse since we learn early that the function will be needed.

The `optimize-js` benchmark results don't exactly reflect the real world. The scripts are loaded synchronously, and the entire parse + compile time is counted towards load time. In a real-world setting, you would likely load scripts using `<script>` tags. That allows Chrome's preloader to discover the script *before* it's evaluated, and to download, parse, and compile the script without blocking the main thread. Everything that we decide to eagerly compile is automatically compiled off the main thread and should only minimally count towards startup. Running with off-the-main-thread script compilation magnifies the impact of using PIFEs.

There is still a cost though, especially a memory cost, so it's not a good idea to eagerly compile everything:



Eagerly compiling *all* JavaScript comes at a significant memory cost.

While adding parentheses around functions you need during startup is a good idea (e.g., based on profiling startup), using a package like `optimize-js` that applies simple static heuristics is not a great idea. It for example assumes that a function will be called during startup if it's an argument to a function call. If such a function implements an entire module that's only needed much later, however, you end up compiling too much. Over-eagerly compilation is bad for performance: V8 without lazy compilation significantly regresses load time. Additionally, some of the benefits of `optimize-js` come from issues with UglifyJS and other minifiers which remove parentheses from PIFEs that aren't IIFEs, removing useful hints that could have been applied to e.g., Universal Module Definition-style modules. This is likely an issue that minifiers should fix to get the maximum performance on browsers that eagerly compile PIFEs.

Conclusions

Lazy parsing speeds up startup and reduces memory overhead of applications that ship more code than they need. Being able to properly track variable declarations and references in the preparser is necessary to be able to preparse both correctly (per the spec) and quickly. Allocating variables in the preparser also allows us to serialize variable allocation information for later use in the parser so we can avoid having to re-preparse inner functions altogether, avoiding non-linear parse behavior of deeply nested functions.

PIFEs that can be recognized by the parser avoid initial preparse overhead for code that's needed immediately during startup. Careful profile-guided use of PIFEs, or use by packers, can provide a useful cold startup speed bump. Nevertheless, needlessly wrapping functions in parentheses to trigger this heuristic should be avoided since it causes more code to be eagerly compiled, resulting in worse startup performance and increased memory usage.

---

1. For memory reasons, V8 flushes bytecode when it's unused for a while. If the code ends up being needed again later on, we reparse and compile it again. Since we allow the variable metadata to die during compilation, that causes a reparse of inner functions upon lazy recompilation. At that point we recreate the metadata for its inner functions though, so we don't need to re-preparse inner functions of its inner functions again. ↩

2. PIFEs can also be thought of as profile-informed function expressions. ↩

Posted by Toon Verwaest (@tverwaes) and Marja Hölttä (@marjakh), sparser parsers.

Retweet this article!