ES2015 and beyond performance plan

Attention: Shared Google-externally

Authors: adamk@chromium.org, bmeurer@chromium.org

Last updated: 2018/10/31

This document describes the proposed plan to improve the performance of new features introduced in ES2015 and beyond. There are a couple of features that are fine by default, because they are merely syntactic sugar for an already optimized feature, but there are quite a few new features that need a separate optimization plan. This document mostly serves as a central place to point to relevant tracking bugs and design documents, it's intended to be a living document so it must be kept up-to-date as we go and address the various issues.

Classes

Owner: bmeurer@chromium.org
Tracking bug: https://bugs.chromium.org/p/v8/issues/detail?id=5455

The design document outlines a couple of ideas how to improve the general performance of classes and related operations.

Class literals

Status: Started
Owner: ishell@chromium.org
Tracking bug: https://bugs.chromium.org/p/v8/issues/detail?id=5799

Class literals go through TurboFan, but cannot really be optimized at this point. It's unclear what the impact of this would be.

Subclassing

Status: **Done**
Owner: tebbi@chromium.org
Tracking bug: https://bugs.chromium.org/p/v8/issues/detail?id=6237

Currently TurboFan cannot inline derived class constructors due to a bug in the inlining logic, which needs proper handling. This should be fixed soon. Besides that, we don't yet have **`new.target`** feedback for leaf constructors, making it impossible to properly optimize the `JSCreate` nodes when they are not inlined. We should consider collecting **`new.target`** feedback on super constructor calls.

Optimizing derived leaf constructors

Status: **Done**
Owner: bmeurer@chromium.org
Tracking bug: https://bugs.chromium.org/p/v8/issues/detail?id=6679 (design document)

Again considering the following example:

```
class A {
  constructor(x) { this.x = x; }
};

class B extends A {
  constructor(x) { super(x); }
};
```

To inline the super call in the `B` constructor we need to add `CallIC` support for `super`, but even then we only have the `JSFunction` constant for `new.target`, which get's us to inlining the `JSCallConstruct`, but we still need some additional magic to be able to also inline the `JSCreate` node.

This could be fixed by tracking the `new.target` as feedback on the `CallIC` for `Construct` and `ConstructWithSpread` bytecodes.

Super property access

Status: Available
Contact(s): bmeurer@chromium.org, adamk@chromium.org

Super property access that doesn't result in a method call doesn't go through the IC system: it's always a runtime call. Unclear what the impact of this is (intuition is that method calls are the majority use case), but it might be nice to avoid a performance cliff here.

Reduce megamorphicity for super class methods

Status: Available
Contact(s): jarin@chromium.org, bmeurer@chromium.org

For super class methods property access to `this` is likely megamorphic, but we could have a smarter way to access the field, since it's usually at the same offset for all subclasses. This would also help with classical class-like ES5 code, for example the TypeScript compiler.

Default derived class constructor (super with spread)

Status: In progress
Owner: petermarshall@chromium.org
Design document: here
Tracking bug: https://bugs.chromium.org/p/v8/issues/detail?id=5659

The default derived constructor desugars to spread/rest, as per the spec. This is extremely slow, however, compared to more straight-forward implementations. This also introduces a hidden performance cliff, as the writer of the JS code can't even see the spread call (and implied iteration). One real-life example of this cliff being hit is in NodeJS.

Collections

Maps & Sets

Status: In progress
Contact: gsathya@chromium.org
Tracking bug: https://bugs.chromium.org/p/v8/issues/detail?id=5717
Design documents: problem statement, SmallOrderedHashTable

Current implementation is written mainly in JS, with special intrinsics like `%_FixedArrayGet`/`%_FixedArraySet` to optimize loading and limit allocations. Still runs into problems with type pollution, though. Rewriting as TurboFan stubs would allow the removal of the scary intrinsics and avoiding the type feedback issues. We should also consider using linear search for small Maps and Sets, and only start hashing once we grow beyond a certain threshold.

One main goal here is to remove the `%_FixedArray` intrinsics/runtime functions that leak internal V8 data structures to JavaScript code, which is very dangerous in general (we've had a couple of serious crashers and security exploit due to leaking internal data structures in the past).

It seems like some popular libraries are already using Maps and Sets heavily (w/ polyfill for older browsers), i.e. see https://twitter.com/sebmarkbage/status/822116812640792576.

{Map,Set}.prototype.forEach

Status: Available
Contact: bmeurer@chromium.org

Currently the forEach methods are implemented in the CSA, but TurboFan doesn't know anything about them. It should be fairly easy to support an inlined version, especially since Maps and Sets are way simpler than Arrays.

Iterators

Iterators are considered too slow to be usable, in combination with both `for-of` and spread.

Array destructuring

Status: Investigating
Owner: bmeurer@chromium.org
Document: bit.ly/array-destructuring-for-multi-value-returns

This came up recently in light of React hook announcement.

Array and String iterators

Status: **Done**
Owner: bmeurer@chromium.org
Tracking bug: https://bugs.chromium.org/p/v8/issues/detail?id=5388 (Array and String iterators)
Implementor(s): caitp@igalia.com

The design document outlines a plan for Array and String iterators. The implementation of String iterators is mostly done (and optimized) already, thanks to Caitlin Potter. The initial version (without escape analysis in TurboFan) already gave a **1.7x** speedup for the string iteration (see results here).

The Array iterators proposal was implemented by Caitlin Potter, and have huge speed-ups coupled with TurboFan's Escape Analysis

Collection iterators

Status: **Done**
Owner: bmeurer@chromium.org
Tracking bug: crbug.com/v8/6571 (Map and Set iterators)

The design document outlines a plan for both Map and Set iterators. The implementation follows along the lines of the ideas used to boost the Array iterator implementation in the past. It's mostly independent of the redesign of Maps and Sets.

for-of

Status: **Done**
Owner: bmeurer@chromium.org
Tracking bug: https://bugs.chromium.org/p/v8/issues/detail?id=3822 (overall for-of performance)

TurboFan can optimize for-of and inline `@@iterator` and `next` function calls; needs more work on the escape analysis and maybe some love for the `try-finally`.

Eliminate the iterator/iterator result allocations

Status: In progress (part of the escape analysis work)
Owner: tebbi@chromium.org
Tracking bug: https://bugs.chromium.org/p/v8/issues/detail?id=5448

TurboFan should be able to completely eliminate the allocations for both the iterator (less important) and the iterator result objects (very important), when the calls to `@@iterator` and `next` are being inlined into the function.

Generators

Status: In progress
Tracking bug: https://bugs.chromium.org/p/v8/issues/detail?id=6351
Contact: adamk@chromium.org

Make generators optimizable

Status: **Done**
Owners: neis@chromium.org, bmeurer@chromium.org

Starting with V8 5.6 generators will go through Ignition+TurboFan only and thus will be fully optimized by TurboFan. Once we have this running in the wild we might want to check the performance again.

Optimize `JSGeneratorObject` creation.

Status: **Done**
Tracking bug: https://bugs.chromium.org/p/v8/issues/detail?id=6352
Contact: mvstanton@chromium.org

Currently the `JSGeneratorObject`s are always created via a call to the `%CreateJSGeneratorObject` runtime function, which is not really efficient.

Async/await

Status: Under investigation.
Tracking bug: https://bugs.chromium.org/p/v8/issues/detail?id=5639
Owners: jgruber@chromium.org, bmeurer@chromium.org
Design document: here (not a design document yet, collecting ideas for now)

Async functions are desugared to generators and thus will be optimized in V8 5.6 as well. There may be a bunch of things we can do on top of the general generator mechanism in Ignition and especially in TurboFan to squeeze better performance out of those (ping bmeurer@chromium.org if you're interested in investigating this). Some ideas:

- Create a version of PerformPromiseThen which does not involve allocating an unused return value (see https://github.com/tc39/ecma262/issues/694, although this could be done without spec changes too).
- Consider if there are cases where the promise machinery can be entirely omitted in favor of just scheduling a microtask to get the right timing. These cases might not be interesting in the real world though (e.g. `await 1`).

This is also closely related to the work on Promises (ping gsathya@chromium.org for details on the Promise work).

In order to make sure we address the right issues, we need to come up with micro and macro benchmarks.

Promises

Status: **Done**
Owner(s): gsathya@chromium.org, adamk@chromium.org
Tracking bug: https://bugs.chromium.org/p/v8/issues/detail?id=5343

JavaScript implementation has had optimization work done on it in Q2 and Q3, driven by the Bluebird benchmark. Much low-hanging fruit was identified, as were cases where we were allocating more things than necessary. A rewrite using the CodeStubAssembler is now complete.

RegExp

Status: **Done**
Owner(s): bmeurer@chromium.org, jgruber@chromium.org, yangguo@chromium.org
Tracking bug: https://bugs.chromium.org/p/v8/issues/detail?id=5339

The RegExp implementation is currently being rewritten to recover the performance loss that we had to take to ship RegExp subclassing initially with the JavaScript builtin based implementation. The new implementation should enable us to add more features w/o regressing performance of older features, and even improve the performance of the existing features (design document).

Modules

Design document (needs updating to match implementation)
Tracking bug: https://bugs.chromium.org/p/v8/issues/detail?id=1569

Variable access

Status: In progress
Owner(s): adamk@chromium.org, neis@chromium.org

Initial implementation uses runtime calls which do dictionary lookups for import/export variable loads/stores. But storage of each variable is in an individual Cell, and the plan is to add IC code to handle these loads/stores. More design necessary, will likely pull in verwaest, ishell, bmeurer.

Prescanning for module requests

Status: Available
Owner(s): adamk@chromium.org, neis@chromium.org

Current API requires modules to be fully parsed before embedder can determine which other modules need to be fetched. Ideally we should be able to notify the embedder while parsing that we've hit a module specifier, allowing preloading of resources in parallel with parsing (analogous to how the HTML parser handles prescanning of URLs in HTML). This is is likely to happen only after Blink fully implements `<script type="module">`.

Arrow functions

Status: Needs TurboFan investigation
Owner(s): bmeurer@chromium.org, mstarzinger@chromium.org
Design document: here (general TurboFan inlining)

Arrow functions are a neat way to create closures; all the optimizations that apply to normal functions also apply to arrow functions. However for arrow functions the most common use case is probably passing that to some other function, thus we should really make inlining based on `SharedFunctionInfo`+`LiteralsArray`+`NativeContext` work, where we are currently limited to inline only concrete closures. The same applies to other short-living closures in general, i.e. also using function syntax.

Short living closures are never optimized

Status: **Done**
Owner: leszeks@chromium.org
Tracking bug: https://bugs.chromium.org/p/v8/issues/detail?id=5512

Short running closures that are called only once will not get optimized at all in V8, unless they contain a loop into which we OSR. This is pretty bad for closure based programming and gets worse with arrow functions as a neat syntax for that.

Spread operator

Status: In progress
Contact: bmeurer@chromium.org

The … operator is currently desugared in the `Parser`, but not in a really efficient way. Investing some time here might give us a pretty big boost in performance (we seem to be orders of magnitude slower in most cases, compared to a very naive ES5 version). This is a main blocker for adoption in the wild.

Spread calls

Status: In progress
Owner: petermarshall@chromium.org
Tracking bug: https://bugs.chromium.org/p/v8/issues/detail?id=5511

There's an investigation and a design document for spread calls.

Array spreads

Status: Needs investigation
Owner: petermarshall@chromium.org
Tracking bug: https://bugs.chromium.org/p/v8/issues/detail?id=5940

See the Design Document for more details.

Currently array spreads in the form `[a,...b]` desugar to an inline for-of using `%AppendElement` to actually append the elements to the array. This is fairly inefficient and cannot be optimized really. Ideally we'd use a `KEYED_STORE_IC` for this, but there's currently no way to tell it not to lookup in the prototype chain. So either unify that, or introduce a custom IC like we did for computed property names.

One option would be to introduce an `APPEND_ELEMENT_IC` that deals with exactly this case, which is fairly simple as it only deals with `JSArray` and only appends.

Let and const

Status: Needs investigation
Owner: adamk@chromium.org
Tracking bug: https://bugs.chromium.org/p/v8/issues/detail?id=5460

Currently `let` and `const` mostly trigger TurboFan, which still seems to be perceived as slower than Crankshaft in many cases. In addition to that there may be some TDZ checks that we don't properly eliminate in TurboFan (or Ignition), so there are probably a couple of low hanging fruits to pick. Additionally, for loops with let or const bindings generate a large amount of AST and extra scopes; there might be wins by teaching the backends how to handle this instead. caitp@chromium.org is investigating this on the frontend side: If a loop variable does not escape from the loop body via capture or eval, and evaluation of the loop body is not suspended by yield or await, then it's *safe* to avoid context allocating these loop variables (and should also be *safe* to avoid the strange desugaring).

Well known symbols

@@hasInstance

Status: **Done**
Owner: bmeurer@chromium.org
Tracking bug: https://bugs.chromium.org/p/v8/issues/detail?id=5640

Currently as soon as anyone installs an `@@hasInstance` property anywhere, we disable basically all optimizations for the `instanceof` operator. That's far more aggressive than it has to be. For example, we could utilize the `GetPropertyStub` to speed up the baseline case of instanceof even when `@@hasInstance` properties are present. Likewise we could inline calls to `@@hasInstance` for `JSInstanceOf` during `JSNativeContextSpecialization` in TurboFan and recognize the special `Function.prototype[@@hasInstance]` builtin in the `JSBuiltinReducer`. This way we don't have this terrible performance cliff with `@@hasInstance`.

Proxies

Status: Assigned

Owners: bmeurer@chromium.org, franzih@chromium.org
Tracking bug: https://bugs.chromium.org/p/v8/issues/detail?id=6557

In general the performance of proxies is perceived as pretty bad in the wild. We could do a lot by at least not leaving JavaScript land for every proxy operation, i.e. utilizing the newly introduced `GetPropertyStub` more often.

[[Call]] and [[Construct]]

Status: **Done**
Owners: bmeurer@chromium.org, franzhi@chromium.org
Tracking bug: https://bugs.chromium.org/p/v8/issues/detail?id=6558

The Call and Construct builtins always go to C++ whenever they hit a `JSProxy` instance. We could easily optimize this utilizing the new `GetPropertyStub` and handle proxies in JavaScript land, which removes the cost of calling back to JavaScript land from C++. This could easily be a factor of 2-3 improvement alone.

[[Get]]

Status: **Done**
Owners: bmeurer@chromium.org, franzih@chromium.org
Tracking bug: https://bugs.chromium.org/p/v8/issues/detail?id=6559

Currently property access via JSProxy objects is handled through C++ runtime functions, which is pretty slow, especially if there's a "get" trap on the handler. With the GetPropertyStub and the CodeStubAssembler technology it should be fairly straight-forward to implement support for [[Get]] on proxy instances w/o going through the C++ runtime (always).

[[Set]]

Status: **Done**
Owners: bmeurer@chromium.org, franzih@chromium.org
Tracking bug: https://bugs.chromium.org/p/v8/issues/detail?id=6560

Same as for [[Get]] above.

Object builtins

Status: Available
Contact: bmeurer@chromium.org
Tracking bug: https://bugs.chromium.org/p/v8/issues/detail?id=5989

The ES2015 and later language revisions added various important builtins on the Object constructor, some of them like `Object.assign`, `Object.create`, or `Object.keys` are used quite heavily in certain use cases. We should do a proper analysis and add missing optimizations of the baseline and even some fast-paths where appropriate.

Object.create

Status: **Done**
Owner(s): bmeurer@chromium.org, cbruni@chromium.org

The baseline implementation is now in the CodeStubAssembler, and there's even support for inlining `Object.create(null)` directly into TurboFan code.

Object.assign

Status: Available
Contact: verwaest@chromium.org
Tracking bug: https://bugs.chromium.org/p/v8/issues/detail?id=5988

The `Object.assign` builtin is very popular in React/Redux applications, where spread properties are commonly used for state management. Babel translates them to `Object.assign` calls. Some fast-path for `Object.assign({},a)` would thus help a lot.

Object.keys

Status: Available
Contact: bmeurer@chromium.org
Tracking bug: https://bugs.chromium.org/p/v8/issues/detail?id=6805

The `Object.keys` built-in already includes a proper fast-path in the `CodeStubAssembler`, but that fast-path still copies the enum cache keys to a new `FixedArray` and allocates a `JSArray`. It might be possible to make the enum cache keys copy-on-write thus avoiding the copying, and avoid the allocation of the `JSArray` by inlining it into TurboFan and letting the escape analysis deal with it.

Object.values and Object.entries

Status: Available
Contact: bmeurer@chromium.org
Tracking bug: https://bugs.chromium.org/p/v8/issues/detail?id=6804

Both the `Object.values` and `Object.entries` built-ins seems to be unnecessarily slow, which might be due to the fact that they are currently implemented in C++, even the fast-path. Maybe porting at least the fast-path to CSA could reduce the **14x** / **2x** slowdown shown in the micro-benchmark in the tracking bug.

Reflect builtins

Status: Available
Contact: bmeurer@chromium.org
Tracking bug: http://crbug.com/v8/5996

With ES2015 we got a whole set of new builtins on the `Reflect` object. Some of these like `Reflect.apply` and `Reflect.construct` already have a fast baseline implementation, and just lack some dedicated fast-paths (i.e. for arguments / rest parameters). Others like `Reflect.set` or `Reflect.has` probably need some investigation for the baseline implementation first.

Reflect.apply and Reflect.construct

Status: **Done**
Owner: bmeurer@chromium.org
Tracking bug: http://crbug.com/v8/5995

We have fast-paths for `Function.prototype.apply` with arguments / rest parameters, and the same fast-paths should be implemented for `Reflect.apply` and `Reflect.construct`. See the relevant part of the design document for details.

Array builtins

Status: Started
Contact: danno@chromium.org, mvstanton@chromium.org
Tracking bug: https://bugs.chromium.org/p/v8/issues/detail?id=1956

See the design document for details.

The performance of various Array builtins like forEach, map and filter is way worse than the performance of a naive ES3 version, I.e. a simple for loop.

Array.from

Status: Investigating
Owner: danno@chromium.org

Tracing bug: https://bugs.chromium.org/p/v8/issues/detail?id=6597

See https://twitter.com/robpalmer2/status/885830194799534081 for the context. The idea is to port the `Array.from` builtin to the `CodeStubAssembler` and make sure that `Array.from(a)` is roughly on-par with `a.slice()` performance-wise.

Inlining higher-order Array builtins

Status: Available
Contact: danno@chromium.org mvstanton@chromium.org
Tracking bug: https://bugs.chromium.org/p/v8/issues/detail?id=2229

See the design document for details.

In order to really achieve peak performance with the `forEach`, `filter`, `map`, etc. builtins, compared to simple for loops, we will need to inline them into TurboFan optimized code. With I+TF launching in M59, this will finally allow us to reach parity.

TypedArrays

Status: Investigating
Owner: bmeurer@chromium.org
Tracking bug: https://bugs.chromium.org/p/v8/issues/detail?id=5929

`TypedArray`s and `ArrayBuffer`s are not well optimized yet. For example we still don't inline TypedArray constructor with (compile time) known arguments. Also the baseline case for the constructors and many utility methods is usually easily pollutable with type feedback because we share the same implementation with all `TypedArray`s and the regular JavaScript arrays and array-like objects. Having dedicated TurboFan builtins here, and thinking about inlinable fast paths would probably help a lot.

The TypedArray subclass constructors

Status: Started
Owner: petermarshall@chromium.org
Bug: https://bugs.chromium.org/p/v8/issues/detail?id=5977

The various typed array constructors are currently written in a weird mix of JavaScript, Crankshaft intrinsics and C++ runtime functions. This is security nightmare and not really efficient either. All of those should be ported to CSA (maybe with some parts just written in C++ completely).

We are starting with porting %TypedArrayInitialize.

Super slow copyWithin builtins

Status: Started
Owner: caitp@chromium.org
Bug: https://bugs.chromium.org/p/v8/issues/detail?id=5925

`TypedArray.prototype.copyWithin` shares the generic code with `Array.prototype.copyWithin` and is thus unnecessarily slow. It should be rewritten as C++ builtin and just call memmove under the hood. That should make it easily 1000-2000x faster. Ported to C++ in https://codereview.chromium.org/2671233002.

TypedArray.prototype.set builtin is slow

Status: Started
Owner: franzih@chromium.org
Bug: https://bugs.chromium.org/p/v8/issues/detail?id=5925 (design document)

Similarly, `TypedArray.prototype.set` is also unnecessarily slow, and could benefit from using `memmove` for the most common cases, see comments on this bug report.

ArrayBuffer.isView is slow

Status: **Done**
Owner: bmeurer@chromium.org
Bug: https://bugs.chromium.org/p/v8/issues/detail?id=6868

The `ArrayBuffer.isView` builtin is used by Node.js core to detect `TypedArray`s and `DataView`s as mentioned in nodejs/node#15663.

TypedArray.prototype[@@toStringTag] is slow

Status: **Done**
Owner: bmeurer@chromium.org
Bug: https://bugs.chromium.org/p/v8/issues/detail?id=6874

The `TypedArray.prototype[Symbol.toStringTag]` getter is currently the best (and as far as I can tell only definitely side-effect free) way to check whether an arbitrary object is a `TypedArray` (either generally `TypedArray` or a specific one like `Uint8Array`). Using the getter is thus emerging as the general pattern to detect `TypedArray`s, even Node.js now adapted it starting with nodejs/node#15663, for the `isTypedArray` and `isUint8Array` type checks.

Destructuring and default parameters

Status: **Done**
Owner: franzih@chromium.org

Destructuring is more or less completely handled within the Parser and can lead to arbitrary large ASTs, which can cause a serious performance hit in some cases, for example a simple one line destructuring can render a (short) function completely un-inlinable. We need to investigate how to reduce the overhead and how we can optimize common destructuring patterns in TurboFan.

Also there's some feedback that destructuring causes deopts. This is caused by the try-catch statement that we must add, which causes Crankshaft to bailout from compilation. Not a problem anymore with TurboFan.

Looking at more specific benchmarks, see https://fhinkel.github.io/six-speed, destructuring is only slow for array patterns, and will improve as `for-of` performances improves.

Rest parameters

Status: **Done**
Owner: bmeurer@chromium.org, tebbi@chromium.org, petermarshall@chromium.org
Design document: here (also covering `arguments` object).

We could look into escape analyzing rest parameters in a bunch of *easy* cases, i.e. when all uses either access the length or the elements. This might be related to optimizations for the (strict) `arguments` object.

Tagged Template Literals

Status: Needs investigation
Contact: bmeurer@chromium.org

Currently, tagged template literals are not handled very efficiently, yet they are very useful in practice. Fully optimizing them to compile away almost all the overhead is probably very difficult, but we there seem to be a couple of low hanging fruits to significantly reduce the overhead. Needs careful investigation.

Computed property names

Status: **Done**
Owner: franzih@chromium.org
Design document: here
Tracking bug: https://bugs.chromium.org/p/v8/issues/detail?id=5622

Currently computed property names in object literals are way slower than adding the property with a dedicated keyed store (which is not semantically equivalent, but the naive way to do it with ES5).

Object rest/spread properties

Status: Available
Contact: bmeurer@chromium.org
Tracking bug: https://bugs.chromium.org/p/v8/issues/detail?id=5789

Additional action items

The Six speed table provides some interesting test cases for performance (rather outdated data), which compares ES6 performance to the performance of the ES5 counterpart.