

The V8 engine


Compilers	3
Introduction	3
Native Compiler vs Cross Compiler	4
Native Compiler	4
What is a native compiler ?	4
How does it work ?	4
What does it do ?	4
Cross Compiler	5
What is a cross compiler ?	5
How does it work 🤖 ?	5
Where is it used ?	5
Summary	5
Performance	5
JavaScript Compilers	6
Introduction	6
Compiler vs Interpreter	6
JIT (Just In Time) vs AOT (Ahead Of Time) compilers	6
Who is the JIT compiler 😬 ?	6
The V8 JavaScript engine	6
But first, what is ignition ?	7
Ok, but what is bytecode ?	7
What do we need ignition for ?	7
And how does ignition work ?	7
Understanding the V8 JavaScript Engine (Step-by-step)	8
Wait, I don't understand how bytecode gets executed !	12
Let's Recap !	13
Apparently, there's more...	13
A little bit of history	13
Maglev	15
Turboshift	16
Garbage collection 🔥	17
Orinoco	17
WebAssembly	18
Introduction	18


Modules	18
Compiler	18
Liftoff	20
Garbage collection (help, get me out of here)	21
Before ending	22
Famous last words	22
The end	22
In depth V8 concepts	23
TurboFan IR (Intermediate Representation)	23
First, some google slides !	23
Edges and Nodes	24
More slides	25
Some actual optimizations	26
TurboFan speculative optimizations	26
Ignition	26
Hidden classes	26
Handlers	27
Inline Caching:	28
Sparkplug	30
JIT-less	30
V8 Irregexp	30

Compilers

"If an image lacks a direct link, it's embedded and accessible with ctrl + click."

" is an important note."

" is an additional note."

" is a reminder note."

Introduction

Compilers translate high-level programming language to low-level programming language.

A compiler is characterized as so:



The **Source** language that is compiled.

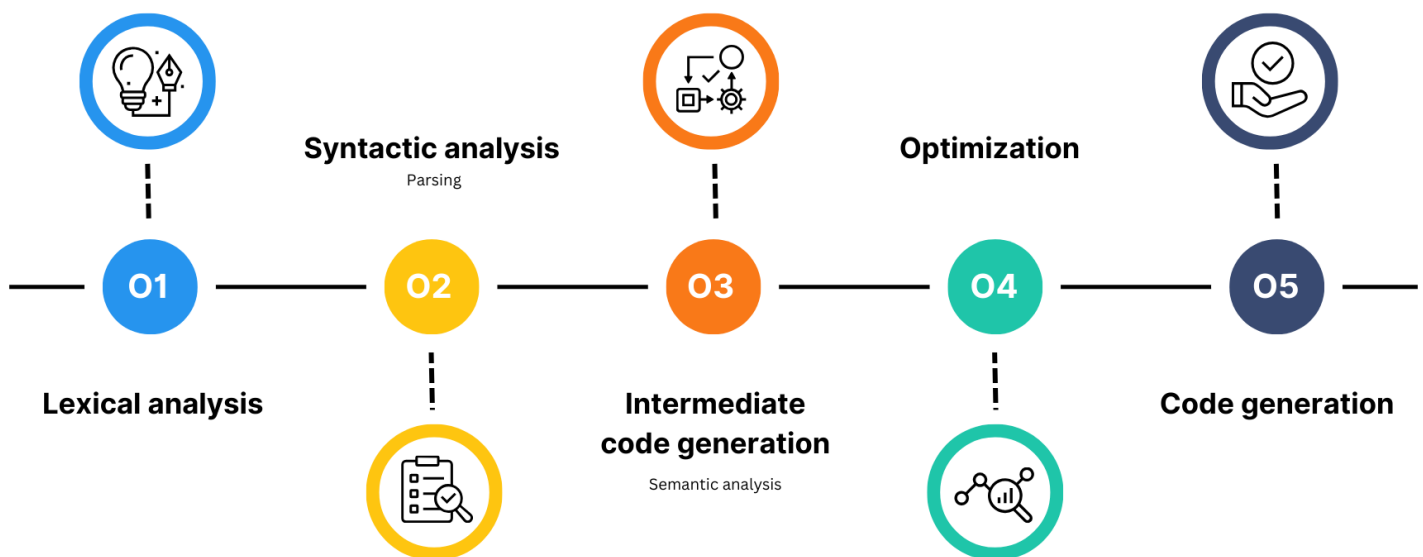


The **Target** language T is generated.



The **Implementation** language that is used for writing the computer.

Compiler phases



Native Compiler vs Cross Compiler

Native Compiler

What is a native compiler ?

Converts high language into computer's native language.

E.g.: if the compiler **runs on a windows** machine, it will **produce executable code for windows**.

- **Easy to set up.**
- **Limited to OS**, thus, not suitable cross-platform development.

How does it work ?

The compiler **analyzes the source code** and then **generates code specific to the OS** of the host machine.

What does it do ?

Various optimizations like **loop unrolling**, **function inlining**, **instruction scheduling**.
Makes code that **execute faster** and more **efficiently**.

Cross Compiler

What is a cross compiler ?

Generates executable code for another platform than the one the compiler is running on.

E.g.: Compiler **runs on a Linux machine** and **produces executable code for Windows**.

- **More difficult to set up.**
- **Multi-platform development.**
- **Not as efficient** as native compilers **in terms of memory usage.**
- Understand the target OS.

How does it work 🤖 ?

The compiler **analyzes the source code** and then **generates code tailored for a different OS.**

Where is it used ?

- In **bootstrapping**, for developing cross-platform software.
- **Microcontrollers**, because they don't support OS.
- **Embedded computers**
- Separates the target environment from the built one.
- ...

Summary

Performance

- Native compilers may **take advantages of the current** platform-specific **optimizations.**
- Cross compilers may be **slower** because the compiled code needs to be **tested and validated on both target platforms.**

JavaScript Compilers

Introduction

Compiler vs Interpreter

1. **JIT compilers generate machine code** at runtime, interpreters don't.
It can **take some time** to compile, **but** execution is then **faster than an interpreter**.
2. The **first compiler in an engine (generate code as fast as possible)** is often called the **baseline JIT compiler**.
3. **Interpreters execute non-compiled code.**
But they don't reach high peak performance.

JIT (Just In Time) vs AOT (Ahead Of Time) compilers

- **AOT:** First **compiles then runs** the code.
- **JIT:** **Dynamically compiles** the code **on-the-fly** during runtime.

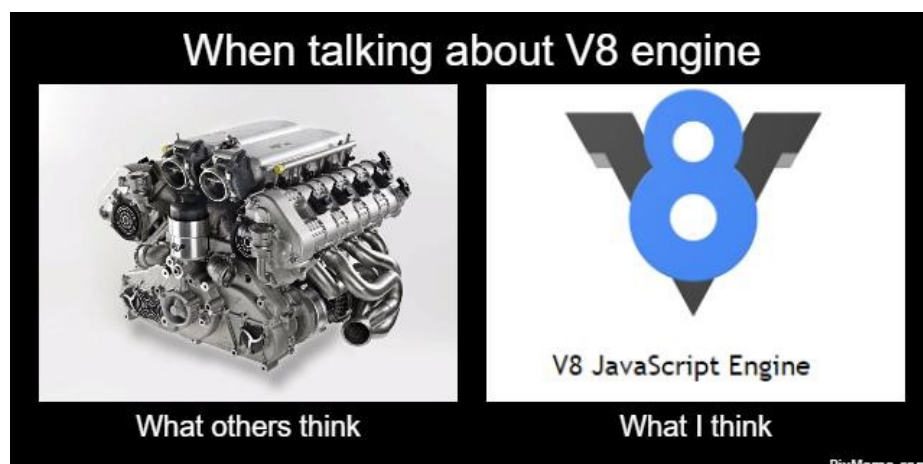
Who is the JIT compiler 🤔 ?

A JIT compiler is **part of the JRE** (Java Runtime Environment) and is **responsible for java applications performance optimization during runtime**.

Another JIT compiler called **Turbofan** is part of the **V8 Google Chrome's engine** which is used by Node.js which is a popular JavaScript's runtime environment.

! Turbofan optimizes and compiles what we call a “hot” function, that's why we call it just-in-time compiler.

The V8 JavaScript engine



But first, what is ignition ?

The name "Ignition" refers to both the **Bytecode Generator** and the **Bytecode Interpreter**.

- ! Ignition is an interpreter.
Even though it "compiles" source code into bytecode, a compiler is considered one if generates machine code.
When someone's refer to a JIT compiler, they usually mean that it's not an interpreter.

Ok, but what is bytecode ?

Bytecode is not a programming language **it is a set of instructions**.

- ! The intent of **Bytecode** is also to be a **portable layer** that gives the software optimal performances.
The intent of **machine code** is to be machine specific and give the **hardware optimal performances**.
Bytecode also executes a lot of different operation to basically wrap up your code, so it becomes ready to be executed by the interpreter.

What do we need ignition for ?

- **Reduce the size of the code-space to** around **50%** of its current size.
- **Have reasonable performances** compared to full-codegen.
- **Full support for DevTools debugging** and CPU profiling.
- A new frontend to the Turbofan's compiler to **enable optimized re-compilation without re-parsing the JS source code**.
- Allow **support for IOS, WASM** (Web Assembly), ...

And how does ignition work ?

Generating

The **Bytecode Generator** **walks** the **AST** (Abstract Syntax Tree) produced by the Parser for the JavaScript source code **and generates bytecode from it**.

- ! **The Parser and the AST** it produces **are not part of Ignition**.

It uses the **BytecodeArrayBuilder** to create the **BytecodeArray** which is the **interpreter input** (where the output is the result of the program).

- ? **It allocates registers** for variables and temporary values, **handles context chains** for nested functions, manages constant pools for storing constants and jump offsets, generates control flow structures using labels and jumps.

Interpreting

The Bytecode Interpreter takes the bytecode generated by the Bytecode Generator and executes it by interpreting it by sending it to a set of Bytecode Handlers.

* see [advanced V8's concepts – ignition handlers](#) for more details.

- ! It translates bytecode instructions, **accessing variables in registers** and calling helper functions for complex operations.
It utilizes inline caches for efficient property access **and aims to collect type feedback for future optimization.**
- ? Each bytecode handler code snippet handles a specific bytecode and dispatches to the handler for the next bytecode.
- ! **The Bytecode Handlers** that make up the Bytecode Interpreter **are generated using parts of the Turbofan (optimizing compiler) pipeline.**
This **happens at V8 compilation time, not at runtime.**
In other words, you need Turbofan to *build* (parts of) Ignition, but not to *run* Ignition.

Understanding the V8 JavaScript Engine (Step-by-step)



*Video: <https://www.youtube.com/watch?v=xckH5s3UuX4>

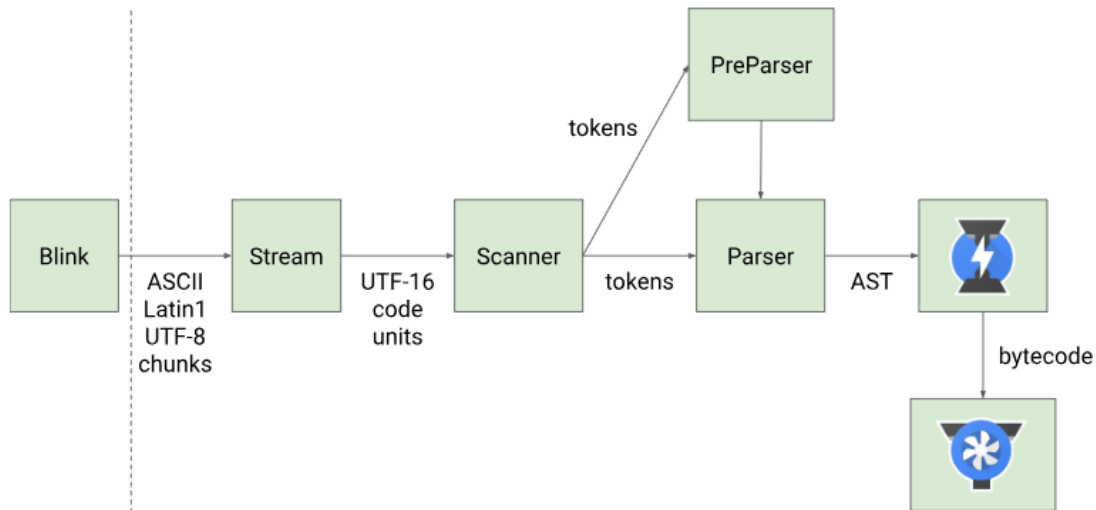
Analysis

1. **Your JS code is broken down into tokens by the scanner** (or lexical analyzer) using various techniques like identifier lookup tables and perfect hashing.
2. **The parser generates the AST (Abstract Syntax Tree) from the tokenized code.**
* you can check any AST generated code here : <https://astexplorer.net/>
3. **The parser also implements something called lazy parsing. Instead of generating an AST for each function, it can decide to “pre-parse” functions instead of fully parsing them.**
It does so by switching to the pre-parser, a copy of the parser **that does the bare minimum needed to be able to otherwise skip over the function.**

When a pre-parsed function is **later called**, it is **fully parsed** and compiled on-demand.

4. **The semantic analyzer checks for source code language correctness and overloaded operators.**

* you can read more about it here: [Why does a compiler need semantics analysis?](#)



Synthesis

5. **Ignition** (low-level register-based interpreter) **bytecode generator generates bytecode from the parsed code.**

! **Just because JavaScript is compiled to bytecode doesn't mean the bytecode codes are compatible across to other languages that also compile to bytecode.**

! E.g., Bytecode used by V8 engine is different from bytecode used in other JavaScript engines.

Other engines names:

Firefox: SpiderMonkey
MSEdge: Chakra
Safari: JavaScriptCore

If the ignition compiled function in **your code**:

1 Is called once:

It is **executed by** the ignition **bytecode interpreter**.

💡 ignition bytecode interpreter executes the generated bytecode by interpreting it via bytecode handlers.

💡 Bytecode handlers are generated using parts of the turbofan low-level, architecture-independent macro-assembly instructions pipeline.

2

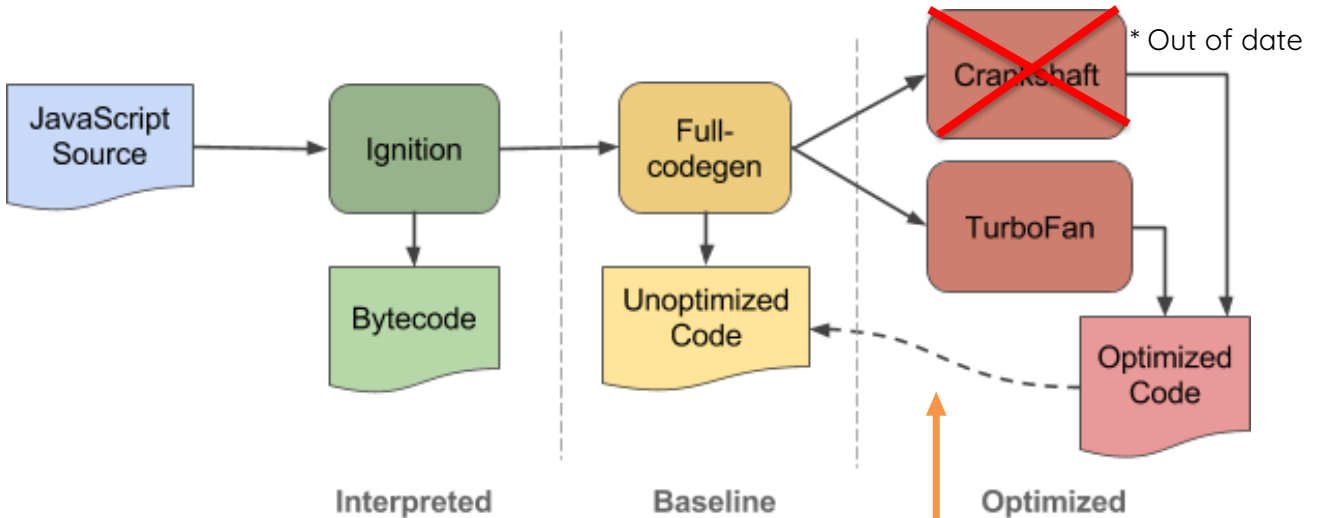
It is **compiled by** sparkplug, a **non-optimizing compiler**.



*Source: [Sparkplug — a non-optimizing JavaScript compiler · V8](#)

3

it is **compiled by** `turbofan`, an **optimizing compiler**, which can now use `sparkplug`'s non-optimized machine code.



*Source: [Firing up the Ignition interpreter · V8](#)
 *Source: [Launching Ignition and TurboFan · V8](#)

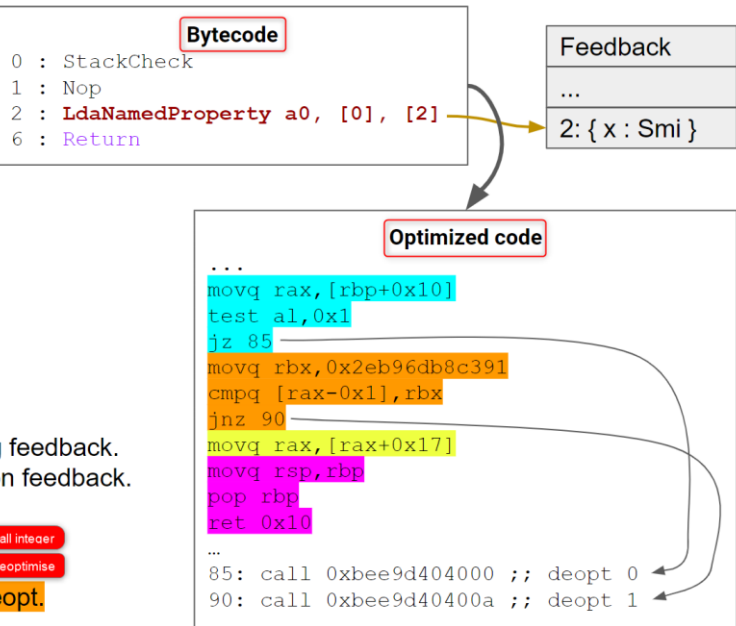
4.1 Deoptimization !??

! Turbofan can also deoptimize code.

Example

```
function f(o) {
  return o.x;
}

f({x : 1}); f({x : 2});
%OptimizeFunctionOnNextCall(f);
f({x : 3});
```



- 1 First, we run unoptimized code, gathering feedback.
- 2 When hot, we decide to optimize based on feedback.
- 3 Optimized code explained:
 1. If o is Smi, deopt.
 2. If o does not have the right map, deopt.
 3. Load o.x.
 4. Tear down frame, return the result.

! When the code deoptimizes immediately it's called **eager deoptimization** (e.g., when the map check fails).

! deoptimizing code happens at runtime.

4.2 Lazy deoptimization

Lazy deoptimization motivating example

```
function change_o() {
  o = { y : 0, x : 1 };
}

var o = { x : 1 };
function f() {
  change_o();
  return o.x;
}

f();
```

```
...
48 call [rdi+0x37] ; Call change_o
;; Lazy deopt point here.
51 movq rax, 0x3415e188941 ;; Get o
61 movq rax, [rax+0x17] ;; Load o.x
65 movq rsp, rbp
68 pop rbp
69 ret 0x8
```

- In this case, change_o will trigger deoptimization of f.
- We cannot deoptimize right away because f is not at the top of the stack.
- We will lazily deoptimize when we return into f.

At the moment we patch all positions after calls with a call to the deoptimizer!

To get a better view on lazy deoptimization consider looking at this fun and easy to understand slides made by Juliana Franco :

[An internship on laziness](#)

6. **Intermediate Representation (IR or intermediate code)** is code that can be compiled to machine code.

It can be :

Three-address code:

It's a **generic assembly language, used and generated by optimizing compilers at compile time.**
It **helps for optimization and code generation.**

Bytecode (or not):

Set of instructions for VM's and compilers generated by a compiler or interpreter and used for execution.

Thus, bytecode could not really be considered as IR because it is not an intermediate step between parsed to machine code.



Remember, bytecode is executed by the interpreter.



Parsed code is compiled to Bytecode and bytecode is compiled to IR.



“Although, Bytecode in V8 is also used as input for the optimizing compiler and in that sense as a step on the way from source text to machine code, if you wanted to call it a special form of intermediate representation, you wouldn't technically be wrong. It would be an unusual definition of the term though”.

*Source: <https://stackoverflow.com/questions/52674479/is-this-an-intermediate-representation#:~:text=since%20the%20bytecode,the%20term%20though>.

7. **The profiler analyzes the code at runtime and determines if it needs optimization.**



JavaScript is a dynamically typed language.
Compilers usually uses information about the types and objects at compile time.
But in JS, **types and properties can change during runtime.**
The profiler keeps track of objects (functions, ...) **during execution in order to obtain some feedback and decide about optimization.**

8. Your machine code and bytecode are now executing bLaZiNgLy FaSt !

Wait, I don't understand how bytecode gets executed !

Bytecode can either be:

- **Translated to machine code by a compiler.**
- **Executed virtually by the interpreter.**



Just like machine code gets executed on your machine.
Bytecode gets converted into machine code but inside a VM.



This involves the need of an interpreter called “Register Machine” which is a type of VM.

Stack-based Machines VS Register-based Machines

- **Stack-based Machines:**
After each instruction, the result is always on top of the stack.
- **Register-based Machines:**
A register machine operates on Virtual Registers.
Ignition is a register machine with each bytecode specifying its inputs and outputs as explicit register operands.

Let's Recap !

1. Your code is tokenized and parsed to AST or lazily parsed (pre-parsed).
2. Ignition takes the AST and outputs bytecode.
3. Sparkplug, the baseline compiler, compiles bytecode (if needed) to non-optimized machine code.
4. Turbofan generates the IR (if needed).
5. Based on ignition profiler feedback (profiling information), turbofan will make speculative optimizations.
6. Turbofan decides if the code needs deoptimization or lazy-deoptimization based on the profiling information and its speculative optimization. Otherwise machine code is generated.

Apparently, there's more...

So, while I was writing this documentation, I've come across a fresh new article on V8's blog. It talks about **Maglev**, which is **a new compiler** introduced in Chrome M117, so let's dive into it 😊 !

It's a **new optimizing compiler** that sits **between Sparkplug and TurboFan**. It serves the purpose "of a **fast optimizing compiler that generates good enough code, fast enough**".

A little bit of history

After bytecode generation **during execution, V8 tracks objects, shapes and types where then runtime execution metadata and bytecode are fed into TurboFan** to generate high-performance, often speculative, machine code.

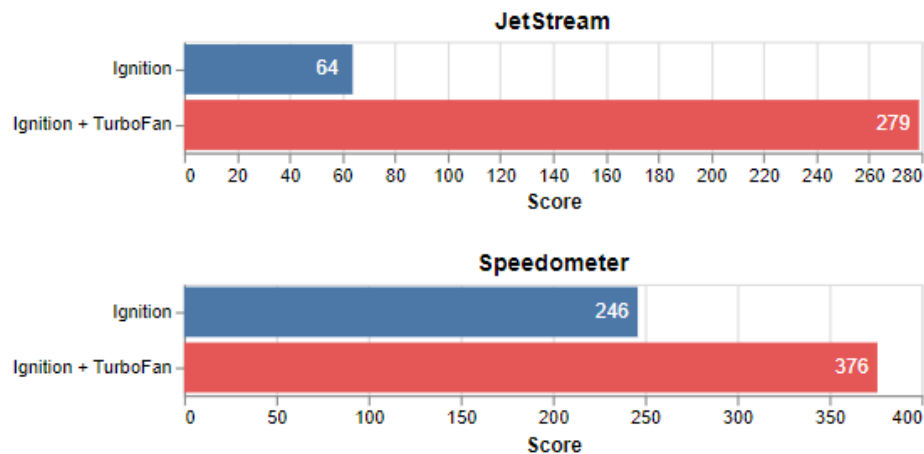
This shows TurboFan **helping V8 running 4.35x** faster than with bytecode only.

The problem

Using a benchmark **measuring startup, latency and peak performance**, the result is **4.35x faster using turbofan**.
(JetStream)

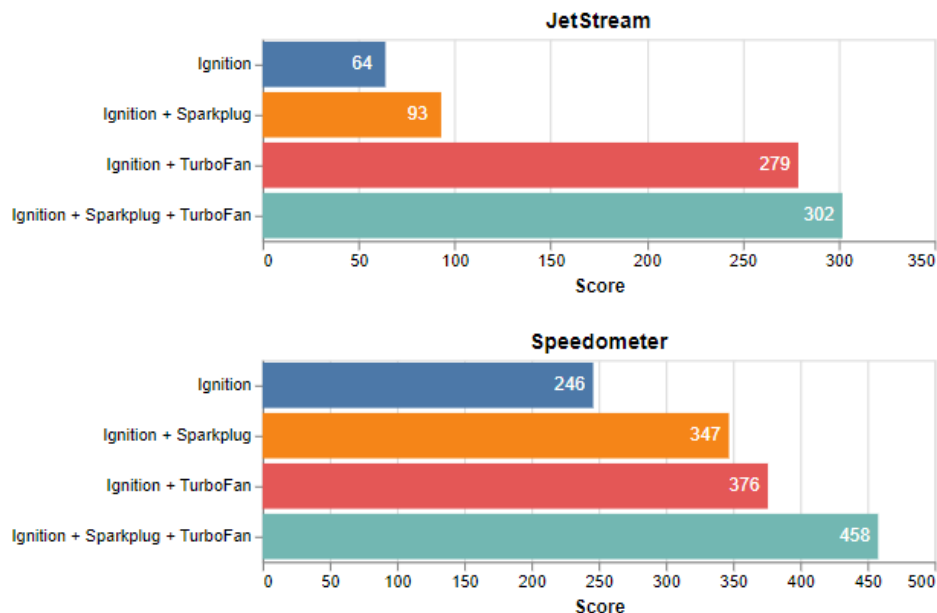
Using another benchmark **measuring web app's responsiveness showed only a 1.5x faster using turbofan**.

This is **due to function that don't get hot enough to be optimized** by turbofan.
(Speedometer)



Improving Ignition

Since the difference in execution speed and compile time between ignition and turbofan is so large, V8 introduced sparkplug.



The problem 2

While **Sparkplug** offers performance improvement over the interpreter, this improvement **has a practical limit due to its simplicity**.

Where, **TurboFan** offers much better potential speedup but **takes longer to compile**.
This creates a gap in performance between Sparkplug and Turbofan.

Maglev

The article explains that “we saw two paths forward to cover the gap between Sparkplug and TurboFan:

- **Either try to generate better code** using the single-pass approach taken by Sparkplug.
- **Or build a JIT with an IR** (intermediate representation).”

Maglev uses a simple, single IR based on SSA (static single-assignment) approach using **CFG** (control flow graph) **rather than Turbofans SON** (sea-of-nodes).

? Fedor Indutny explains in his blog :
“The striking difference between this graph (SON) and CFG is that there is no ordering of the nodes, except the ones that have control dependencies (in other words, the nodes participating in the control flow).”

*Source: [Sea of Nodes \(darkside.de\)](https://darksidede.github.io/sea-of-nodes/)

Why maglev ?

In short, **Maglev** :

- Tries to make TurboFans life easier by **pre-optimizing**.
- It **lowers CPU cost**.
- Allows TurboFan to come a little bit later, thus **increasing responsiveness and loading**.

How it works

1. **Analyzes bytecode** to find branch targets, loop assignments, and liveness information (which values are needed).
2. **Represents expressions and variables** using a single assignment to each variable name.
3. **Merge values from different branches** at control flow points (e.g., loops).
4. **Leverages runtime feedback** to optimize code based on observed object shapes and types.
5. **Deoptimizes** if needed.
6. Chooses the **most efficient representation** for numbers (e.g., integers vs. floating-point).
7. Decides **where to store values** during execution (registers vs. stack).
8. Translates “optimized” **IR into machine code**.

Conclusions ?



- Enables **earlier optimization** compared to TurboFan.
- Compiles code more efficiently (thus, **lower resource usage**).
- Consumes less energy.**
- Better** performance in web page **loading speed and responsiveness.**

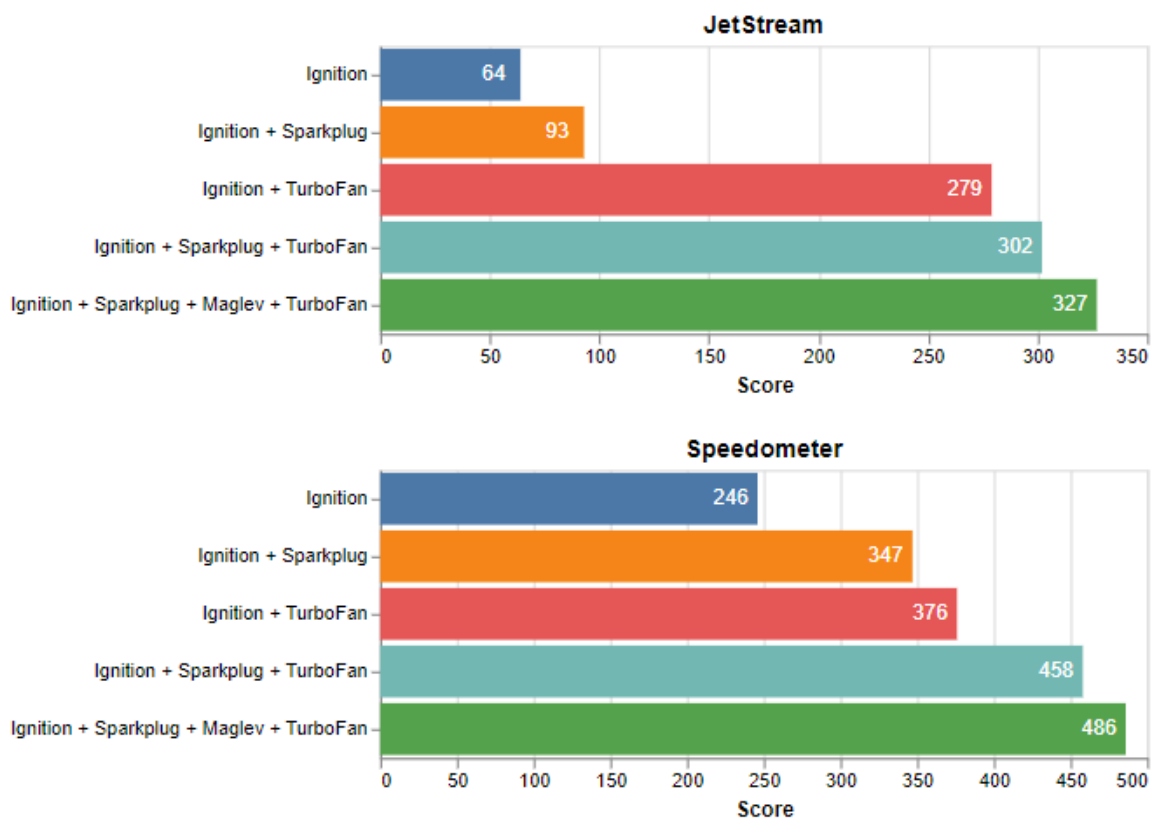
Results

“In terms of **compilation speed**, we’ve managed to build a JIT that’s roughly **10x slower than Sparkplug, and 10x faster than TurboFan**”.

Thus, maglev can be deployed faster than TurboFan.

Also, the cost of deoptimization and recompiling has no huge costs here.

And Maglev allows V8 to use TurboFan a little later, which also adds a lot.



Turboshift

Turboshift is an incoming chapter; it’s currently developed and the new architecture for TurboFan that makes it compile twice as fast.

Garbage collection 🗑️

Orinoco

C/C++ require manual memory management by programmers, increasing the risk of memory leaks and crashes.

Languages like **Java and JavaScript have built-in garbage collectors** that automatically manage memory, freeing programmers from it 😊.

The Problem of Blocking the Main Thread

Traditional **garbage collection can block the main thread** for extended periods, impacting performance.

In a video playing at 60 FPS, a blocked main thread for more than 16.6ms can lead to dropped frames and buffering.

Strategies for Concurrent Garbage Collection

- The **Thread Model distributes workload among multiple threads** to minimize main thread blocking.
- The **Interleaved Model runs GC at smaller intervals** for quicker completion but suffers from increased context switching overhead.
- The **Multi-Processing Model runs GC and the main thread in parallel** on separate cores (ideal but resource-intensive).

Identifying Dead Objects

Orinoco traverses the stack and global pointers to **identify unreachable objects** (considered dead) **and flag reachable objects as live**.

- ❓ Freeing dead object memory leads to external fragmentation. Compaction rearranges memory to consolidate free space and improve allocation efficiency.

Generational Garbage Collection in Orinoco

Orinoco generation hypothesis is based on the fact that **objects created recently are more likely to die young**. Thus, it categorizes objects into young and old generations and **objects surviving a GC cycle are promoted to the old generation**.

Minor GC (Scavenger)

Uses a thread model for fast execution.

Identifies dead objects only in the young generation.

Evacuates live objects to a new, contiguous location to avoid fragmentation.

Updates stack and global pointers to reflect the new location of live objects.

Major GC

Calculates the next GC time slot based on object allocation and survival rates.

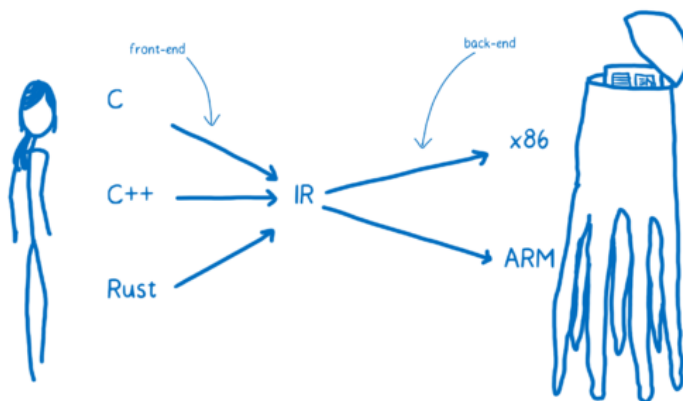
Uses a multi-process model to mark dead objects (across all generations).

Employs a multi-threaded model to free memory occupied by dead objects.

WebAssembly

Introduction

- 🤖 Assembly language acts as an intermediary between high-level programming languages (like JavaScript) and machine code.
- 😓 Different machine architectures have different assembly dialects due to their varying internal structures.
- 🧐 **Compilers translate high-level languages into assembly by using an IR as a middle step. This allows a single front-end of the compiler to handle various high-level languages, while the back-end can specialize in generating assembly for specific architectures.**

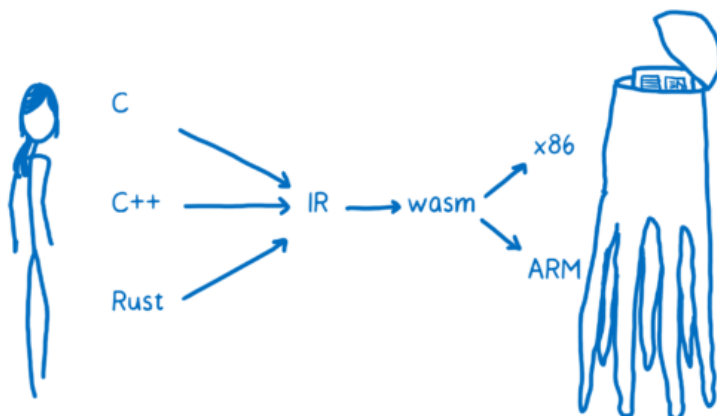


*Source: [A crash course in assembly - Mozilla Hacks - the Web developer blog](#)

Modules

WebAssembly modules define functions that can be used from JavaScript.

WebAssembly is a machine language for a conceptual machine, not an actual, physical machine.



Compiler

The compiler tool chain that currently has the most support for WebAssembly is called LLVM. There are a number of different front-ends and back-ends that can be plugged into LLVM.

From C to WebAssembly

1. We could use **clang front-end to go from C to the LLVM IR**.
2. To **go from LLVM's IR to WebAssembly** we could **use emscripten back-end**.
3. The end result is a file that ends with .wasm.

? WebAssembly can only use numbers as parameters or return values, so you need to play with array buffers to pass anything else.

*Source: [WebAssembly.Memory - WebAssembly / MDN \(mozilla.org\)](https://developer.mozilla.org/en-US/docs/WebAssembly/Memory)

Here is a web tool to compile to wasm: mbebenita.github.io/WasmExplorer/

Stack machine

Remember when we talked about ignition being a register machine ?

→ Right [here](#).

Well, this is because like ignition is a register machine, **WebAssembly is a stack machine**.

? “
Operations like `add` function know how many values they need. Since `add` needs two, it will take two values from the top of the stack. This means that the `add` instruction can be short (a single byte), because the instruction doesn't need to specify source or destination registers. This reduces the size of the .wasm file, which means it takes less time to download.

”

“Physical” machines use registers.


Each machine has a different number and types of registers (x86, ARM).


- When **your browser** encounters Wasm code, it **acts as a translator**.
- It takes the Wasm stack-based instructions and figures out how to achieve the same results using the physical machine's registers.
- **This translation happens “just-in-time” (JIT), meaning it's done on the fly as the code is needed.**
- Since Wasm doesn't specify registers, the browser has the freedom to choose the most efficient register allocation for that particular machine.

Analogy

 Imagine you have a recipe that calls for adding ingredients “one by one.” (This is Wasm)

 You can follow this recipe in any kitchen (different machines) because the steps are clear.

 But a well-equipped kitchen might have bowls and containers (registers) to organize things for faster cooking.

 The cook (browser) decides how to use those tools (registers) based on the recipe (Wasm) and the kitchen setup (machine architecture).

Liftoff

Liftoff is the baseline compiler for WebAssembly in V8.

Why liftoff ?

Liftoff is introduced to improve the startup time of WebAssembly applications. Traditional compilation with TurboFan is time-consuming because it prioritizes code quality over speed.

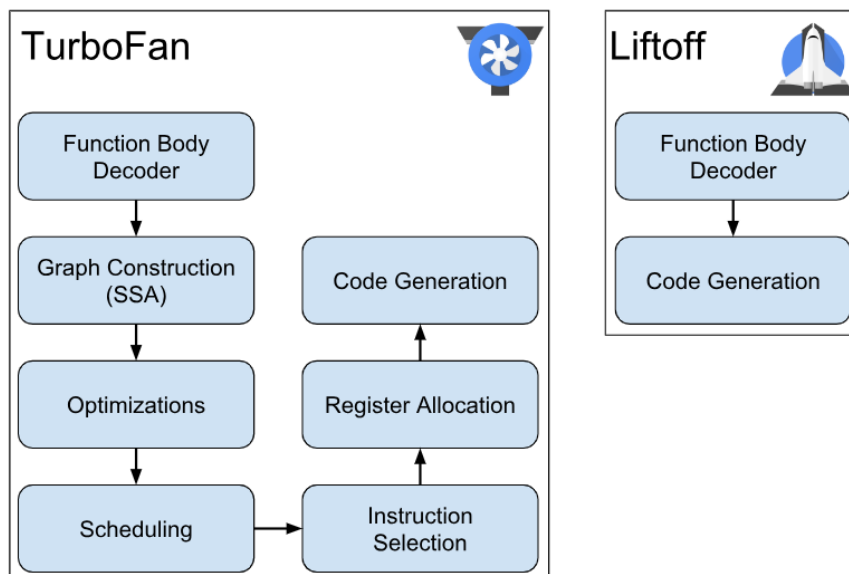
Liftoff on the other hand, generates code quickly at the expense of code quality. Since the generated code is less performant, V8 employs TurboFan to compile an optimized version in the meantime.

This optimized code gradually replaces the Liftoff code, ensuring smooth performance as the application runs.

Understand that liftoff is a one-pass compiler, it iterates over the WebAssembly code once and emits machine code immediately.

Slight Trade-off but fast output 🤔

The trade-off is a slight performance drop during the initial execution phase, before TurboFan's optimized code takes over.



The Liftoff compilation pipeline is much simpler compared to the TurboFan compilation pipeline.

*You can “decompile” wasm using the wabt toolkit as it’s intended to be human more readable:
<https://github.com/WebAssembly/wabt>

Recap ? Yes !

1. When a Wasm function is called for the first time, Liftoff compiles it and generates machine code.
2. If a function becomes “hot,” it triggers TurboFan recompilation on a background thread.
3. TurboFan generates optimized code and replaces the Liftoff code in the Wasm module.
4. Subsequent calls to the function use the optimized TurboFan code.

Garbage collection (help, get me out of here)

There are 2 main approaches :

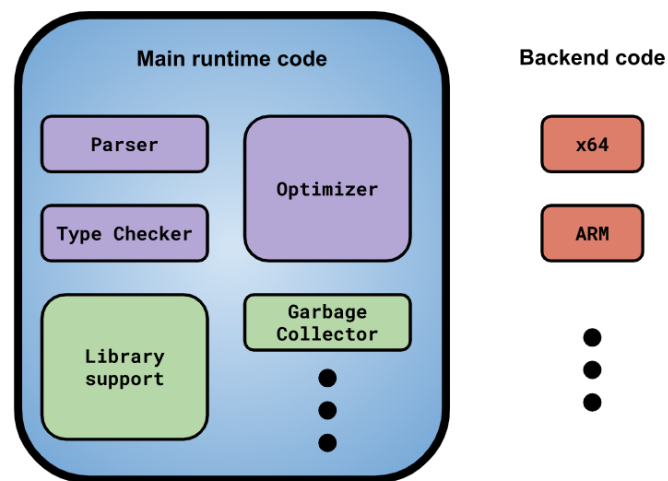
- The “**traditional**” **porting approach**, in which **an existing implementation of the language is compiled to WasmMVP**, that is, the WebAssembly Minimum Viable Product.
- The **WasmGC porting approach**, in which **the language is compiled down to GC constructs in wasm itself** that are defined in the recent [GC proposal](#).

Traditional

The general idea is then to **recompile the VM** to the targeted architecture.

Also, **if the VM has architecture-specific code, like JIT or AOT compilation, then you also implement a backend for JIT/AOT for the new architecture.**

The main part of the codebase can just be recompiled for each new architecture you port to:



Structure of a ported VM

Here, the parser, library support, garbage collector, etc., are all shared between all architectures in the main runtime.

Porting to a new architecture only requires a new backend for it, which is a comparatively small amount of code.

there are several Wasm-specific downsides to this approach, and that is where WasmGC can help.

WasmGC

It allows you to define language constructs using WasmGC primitives (structs & arrays), cast between types, create instances,

Those objects are managed by the wasm VM's own GC.

So, the traditional approach ports a language to an architecture and the WasmGC approach ports a language to a VM.

! **This architecture/VM metaphor is not an exact one**, in particular because WasmGC intends to be lower-level than the other VMs.

! Also, **this eliminates the need to bundle a separate garbage collector** with the compiled language, reducing the size of the Wasm binary.

Conclusion

While traditional porting offers faster implementation due to VM recompilation, **WasmGC provides several advantages in terms of memory management efficiency, developer tool integration, and potential for improved performance.** The trade-off lies in the increased toolchain effort required for WasmGC ports.

Before ending

Here are some cool articles you might want to read:

[Tracing from JS to the DOM and back again · V8](#)

[Introduction to WebAssembly • rsms](#)

[High-performance garbage collection for C++ · V8](#)

Note that we discussed a lot of V8 optimizations, but the first optimization comes from the garbage (collected) code you write. Here are some tips from MDN 📌:

[JavaScript performance optimization - Learn web development | MDN \(mozilla.org\)](#)

Famous last words

Thank you for reading this article. Learning about the V8 engine was really painfu...

HUM interesting 🧐.

More seriously, thank you for reading and if you have any suggestions or comments, please feel absolutely free to contact me on discord, this is my id `positiveblues`.

This documentation took me two and a half months (since I'm working and a student), you can find similar work on my GitHub profile in the [learn-fast](#) repository. If you want to support me and my work you can buy me a coffee on [ko-fi.com/matthiasbrat](#), or just come talk to me on discord 😊 !

Ps: You can find the sources at the end of the document; it doesn't contain all the articles I've read but only the most important ones.

The end

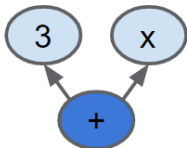


In depth V8 concepts

TurboFan IR (Intermediate Representation)

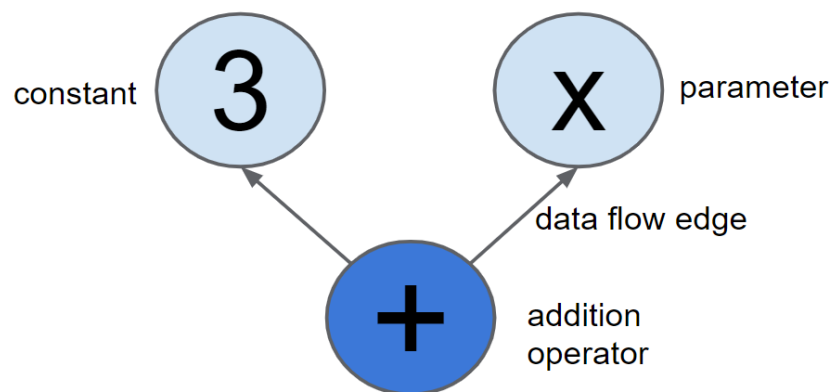
First, some google slides !

Do not get seasick!



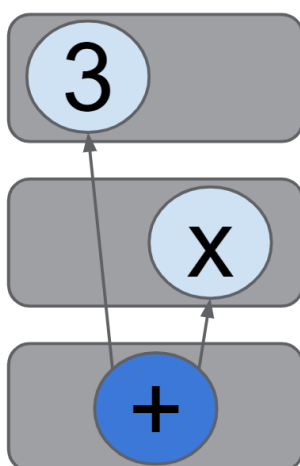
All computations are expressed as nodes in the sea of nodes

Edges represent dependencies between computations

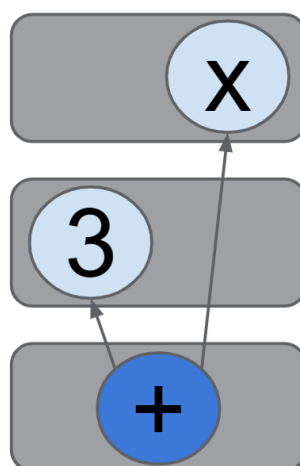


https://docs.google.com/presentation/d/1sOEF4MIF7LeO7uq-uThJSuJITh--wgLeaVibsbb3tc/edit#slide=id.g5499b9c42_0790

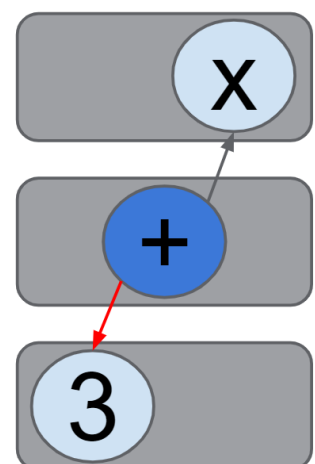
Dependencies constrain Ordering



😊 legal



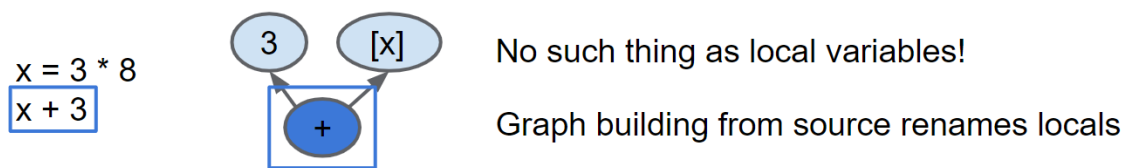
😊 legal



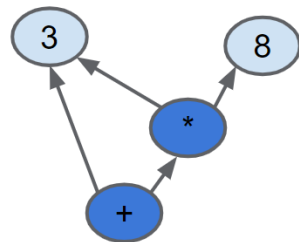
🚫 illegal

https://docs.google.com/presentation/d/1sOEF4MIF7LeO7uq-uThJSuJITh--wgLeaVibsbb3tc/edit#slide=id.g5499b9c42_0836

The Sea is SSA



SSA
renaming



Multiple incoming
edges possible

https://docs.google.com/presentation/d/1sOEF4MIF7LeO7uq-uThJSuJJITh--wgLeaVibsbb3tc/edit#slide=id.g5499b9c42_0816

SSA means “Static Single-Assignment”.

In compiler design, static single assignment form is a property of an intermediate representation (IR) that requires each variable to be assigned exactly once and defined before it is used.


Edges and Nodes

Edges are the relationships between nodes.

There are different **types of edges**:

- **Data flow edges:**
Indicate the dependencies between the values computed by the nodes.
- **Effect edges:**
indicate the order of execution of the nodes that have side effects.
- **Control edges:**
indicate the conditional branches of the program flow.
You can think of edges as the **glue** that connects the nodes in the sea of nodes representation.


Nodes are the basic units of computation in the sea of nodes graph.

 **Each node represents an operation** (constant, parameter, arithmetic expression, call...).

 **Nodes have inputs and outputs, which are connected by edges to other nodes.**

 Nodes can have different **types of outputs**:

- **Value outputs:**
The result of the computation.
- **Effect outputs:**
The side effects of the computation.
- **control outputs:**
The conditional branches of the computation.

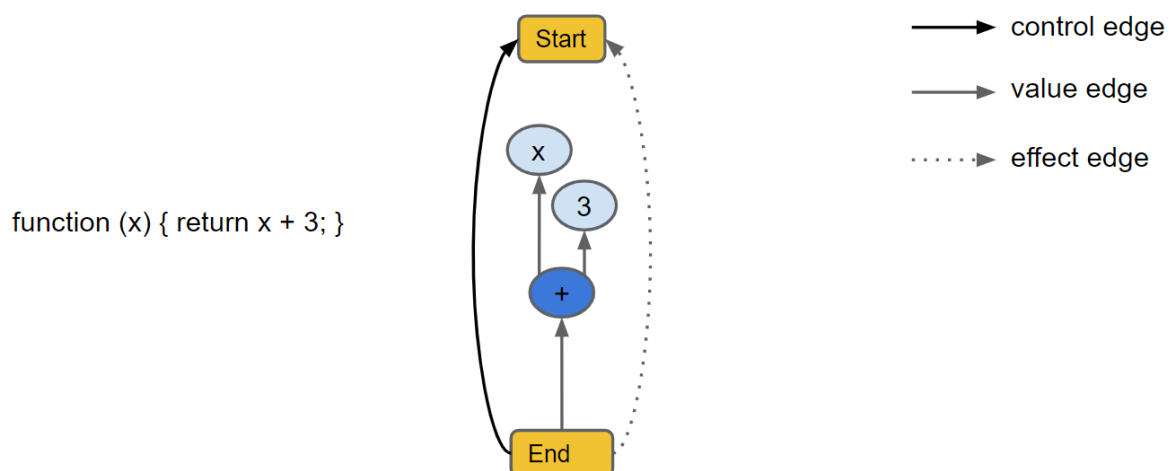
 Nodes can also have **different shapes**:

- **Circle node - constant:**
Represents a fixed value that does not depend on any other node.
- **Diamond node - conditional:**
Represents a branch in the program flow based on a condition.
- **Rectangle node - side effecting:**
Represents an operation that changes the state of the program or the environment.

* On these slides, we have constant, conditional and side effecting nodes represented with different shapes and characterized by different colors .

More slides

Our first complete graph



https://docs.google.com/presentation/d/1sOEF4MIF7LeO7uq-uThJSuJIJTh--wgLeaVibsbb3tc/edit#slide=id.g5499b9c42_01215

Language Levels

- JavaScript:
 - JSAdd JSSubtract JSMultiply JSDivide JSModulus JSDivide
 - JSBitwiseOr JSBitwiseAnd JSBitwiseXor JSShiftLeft JSShiftRight
 - JSEqual JSStrictEqual JSToBoolean JSToNumber JSCall
- Intermediate:
 - NumberAdd NumberSub NumberMul NumberDiv NumberMod
 - NumberEqual NumberLessThan LoadField StoreField
 - StringEqual StringAdd ChangeTaggedToInt32
- Machine:
 - Int32Add Int32Sub Int32Mul Float64Add Float64Sub Float64Mul
 - Load Store Call ConvertFloat64ToInt32

https://docs.google.com/presentation/d/1sOEF4MIF7LeO7uq-uThJSuJJITh--wqLeaVibsbb3tc/edit#slide=id.g5499b9c42_01170

Some actual optimizations

From now on you should be able to read the rest of the google Google Slides presentation where you will find the list of the actual optimization handled by using the sea of nodes.

[TurboFan TechTalk presentation - slide 13](#)

TurboFan speculative optimizations

Turbofan also receives ignition feedback and makes assumptions based on feedback and SON which are called speculative optimizations.

Turbofans type guards will then check if the shape of the object is correct, and if not, its deoptimized back to bytecode.

Ignition

Hidden classes

- **Hidden classes:**

Every object in V8 is associated with a unique hidden class (also called "shape" or "map"). While each object has its own class, **objects with the same property structure can share the same hidden class**, optimizing memory usage.

It tracks the layout of objects, allowing ignition to quickly access properties and optimize code performance.

- **Hidden class transitions:**

Allows to share information between old and new shapes for similar objects.

Hidden class transitions form a tree-like structure, with objects pointing to different nodes based on their property history.

```
var obj1 = {}; // Empty hidden class
obj1.a = 1; // New hidden class with "a" offset
var obj2 = {}; // Empty hidden class
obj2.b = 2; // New hidden class with "b" offset
obj2.a = 1; // New hidden class with "a" and "b" offsets (different from obj1)

// obj1 and obj2 have different hidden classes despite having the same properties
// due to their distinct assignment order.
```

Handlers

- **Individual code objects:**

Each handler is generated independently as its own TurboFan code object.

- **Assembler:**

A specialized assembler generates handlers using high-level primitives like Dispatch and GetBytecodeOperand.

- **Not directly called:**

Handlers only dispatch to the next handler using tail calls.

- **State management:**

Fixed registers hold interpreter state (BytecodeArray, offset, accumulator) across handlers.

- **Parameter passing:**

These state values are treated as parameters passed through dispatches without stack operations.

- **Simplified pipeline:**

The generated handler graph goes through a simplified TurboFan pipeline for optimization.

Entry and Stack Setup:

1. The InterpreterEntryTrampoline builtin sets up the stack frame and initializes interpreter registers.
2. It allocates space for the register file and initializes all registers to undefined (for GC safety).
3. Fixed registers hold the bytecode pointer, dispatch table pointer, and other interpreter state.

4. The first bytecode handler is called.

Register Access:

1. Variables and temporaries are assigned registers during bytecode generation.
2. Bytecode handlers use register operands to specify which register to access.
3. The interpreter scales the operand index to calculate the byte offset within the register file.

Bytecode Opcodes and Dispatch:

** opcode, or operation code, tells the ALU what operation to perform.*

- Ignition uses prefix bytecodes (Wide and ExtraWide) to increase operand size while minimizing bytecode size.
- The dispatch table has entries for different operand sizes to handle prefixed bytecodes.

JS Function Calls:

1. The Call bytecode handles calls to other JS functions.
2. It specifies the callee, argument registers, and number of arguments.
3. The interpreter pushes arguments onto the stack and calls the InterpreterPushArgsAndCall builtin.

This builtin uses the same Call builtin as full-codegen for optimized function calls.
4. When returning, the interpreter tail calls the InterpreterExitTrampoline builtin to return control.

Inline Caching:

It leverages the **assumption that functions are often called with the same type of object**, especially within loops or heavily used code sections.

- Bytecodes use inline caches (ICs) to load and store properties on objects.
- The same ICs used by full-codegen are used by Ignition, allowing shared type feedback collection.
- **Optimistic Assumptions:**
Ignition makes assumptions about future code behavior based on past execution using inline caching (ICs) and memory offsets, speeding up interpretation but potentially requiring deoptimization if those assumptions become incorrect.

- **Monomorphic IC (Optimized):**

When the function is always called with objects of **exactly the same type**, the caching is monomorphic, leading to the most significant performance improvement.

```
function greetPerson(person) {
    console.log("Hello, " + person.name + "!");
}

const person = { name: "Alice" };

for (let i = 0; i < 1000; i++) {
    greetPerson(person); // Always called with the same object
}
```

- **Polymorphic IC (Slightly Optimized):**

If the function is called with a **limited number of different object types**, the caching is polymorphic, offering some level of optimization compared to the unoptimized case.

```
function processData(data) {
    if (data.type === "number") {
        console.log("Number:", data.value);
    } else if (data.type === "string") {
        console.log("String:", data.value);
    } else {
        console.log("Unknown data type");
    }
}

const data1 = { type: "number", value: 42 };
const data2 = { type: "string", value: "Hello" };

for (let i = 0; i < 100; i++) {
    processData(data1);
    processData(data2);
}
```

- **Megamorphic IC (Unoptimized):**

If the function is called with a **wide variety of object types**, the caching becomes megamorphic, and no specific type information is stored, resulting in no optimization benefit from inline caching.

```
function handleInput(input) {
    console.log(input);
}

const inputs = [
    { type: "number", value: 10 },
    { type: "string", value: "text" },
    { name: "Alice", age: 30 }, // Different structure
];

for (const input of inputs) {
    handleInput(input);
}
```

Sparkplug

- “ In fact, Sparkplug code is basically just builtin calls and control flow. ”
- “ it removes (or more precisely, pre-compiles) those unremovable interpreter overheads, like operand decoding and next-bytecode dispatch. ”
- “ Sparkplug is a “transpiler” from Ignition bytecode to CPU bytecode, moving your functions from running in an “emulator” to running “native”. ”

JIT-less

In V8 version 7.4, the **devs introduced a JIT-less flag** that allows you to **make your code interpreted by ignition ONLY**.

Which is surprising at first because “V8 relies heavily on the ability to allocate executable memory at runtime”.

What makes V8 fast is that :

- TurboFan makes just-in-time native code for “hot functions”.
- Regexp is passed through the Irregexp engine.

Both needed for creation of executable memory at runtime.

V8 Irregexp

Actually, V8 also has a **RegExp engine called Irregexp**.

It's **built on V8 existing infrastructure** for memory management and native code generation.

The problem

As improvements went on the V8 engine, **RegExp started to stand out as being slower than the rest**.

Irregexp first converts a RegExp into an **intermediate automaton representation** (which is a very accessible and easy to understand representation).

After optimization, the engine generates machine code which **was** used with backtracking to try different alternatives **until V8 version 8.8**, which led to performance issues.

- ❓ Backtracking occurs when the engine explores alternative interpretations of the pattern, potentially revisiting the same positions repeatedly in the input string.)

How it works

This new engine employs a **non-backtracking algorithm** inspired by **breadth-first traversal** of the automaton representing the RegExp.

It considers all possible interpretations simultaneously, **eliminating duplicates and advancing states based on the current input character**.

This approach guarantees **linear time complexity** with respect to the input string length for a large class of patterns.

This happens because:

“

1. Some platforms (e.g. iOS, smart TVs, game consoles) prohibit write access to executable memory for non-privileged applications, and it has thus been impossible to use V8 there so far; and
2. disallowing writes to executable memory reduces the attack surface of the application for exploits.

”

<u>Website Origin</u>	<u>Source</u>
GeekforGeeks	Difference between Native compiler and Cross compiler What is Cross Compiler? Instruction Level Parallelism
Wikipedia	Java bytecode Three-address code
V8 – Google Docs	Ignition Design Doc TurboFan TechTalk presentation Turbofan IR Deoptimization in V8 An internship on laziness
V8	Ignition TurboFan Launching Ignition and TurboFan Blazingly fast parsing, part 2: lazy parsing Maglev - V8's Fastest Optimizing JIT An additional non-backtracking RegExp engine Liftoff: a new baseline compiler for WebAssembly in V8 What's in that .wasm? Introducing: wasm-decompile

	A new way to bring garbage collected programming languages efficiently to WebAssembly WebAssembly compilation pipeline
Stack Overflow	What does V8's ignition really do? The confusion with JIT compilation in V8
Quora	What does byte code look like? How does it differ from assembly language? What's the difference between Node.js and Java at the byte code level when both are run a virtual machine? Why does a compiler need semantics analysis?
YouTube	Understanding the V8 JavaScript Engine - Lydia Hallie How Does JavaScript Work ?
GitHub	P0lip/v8-deoptimize-reasons yanguly/v8-playground
Medium	Sneak peek into Javascript V8 Engine by Pooja Sharma 灰豆 Huidou Inside JavaScript Engines, Part 2: code generation and basic optimizations by Yan Hulyi
Else	https://astexplorer.net/ http://web.archive.org/web/20151010192637/http://www.dound.com/courses/cs143/handouts/17-TAC-Examples.pdf https://blog.chromium.org/2010/12/new-crankshaft-for-v8.html https://chromium.googlesource.com/v8/v8/+refs/heads/main/tools/turbolizer/ Sea of Nodes (darks.de) https://lampwww.epfl.ch/teaching/archive/advanced_compiler/2008/resources/slides/acc-2008-12-ssa-form_6.pdf https://blog.chromium.org/2009/02/irregexp-google-chromes-new-regexp.html https://hacks.mozilla.org/2017/02/a-crash-course-in-assembly/ https://hacks.mozilla.org/2017/02/creating-and-working-with-webassembly-modules/ https://mbebenita.github.io/WasmExplorer/ https://developer.mozilla.org/en-US/docs/WebAssembly/JavaScript_interface/Memory https://developer.chrome.com/blog/wasmgc

Made by :
Matthias Brat

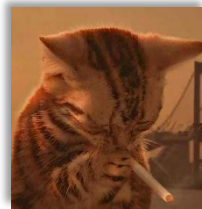
Socials:



github.com/matthiasbrat



stackoverflow.com/users/17921879/matthias-b



matthiasbrat.dev