FAST FAST FAST : JavaScript vs C++

Prasanna Patil
prasannapatil038@gmail.com

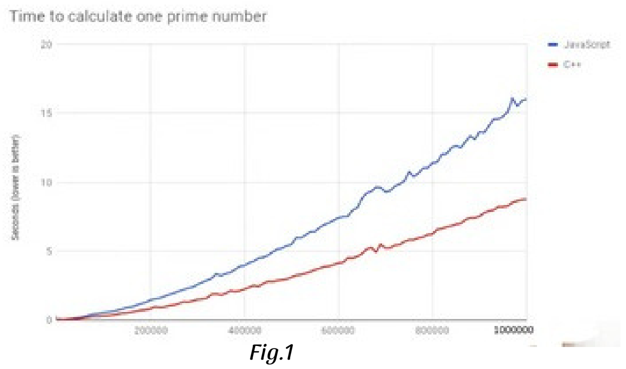

Fig.1

JavaScript belongs to a category of scripting languages, which are designed to be **interpreted rather than compiled** directly into machine code before execution. Unlike statically typed languages like C++, JavaScript is **dynamically typed**, meaning variables do not have fixed data types at compile time. This flexibility makes JavaScript highly adaptable but also presents challenges in optimizing performance. Other examples of scripting languages include **PHP (Hypertext Preprocessor)**, which is widely used for server-side web development. JavaScript, however, is unique because it runs inside web browsers and relies on **Just-In-Time (JIT) compilation techniques** to boost execution speed.

When discussing JavaScript's performance, it's crucial to clarify that we are focusing strictly on **runtime execution speed**—how fast JavaScript executes code in memory. This means we are not considering external factors like **file operations, database queries, or network I/O**, which depend on entirely different optimizations. Instead, this article delves into how JavaScript engines process and execute code efficiently, particularly through techniques like baseline and optimizing compilers.

To demonstrate JavaScript's efficiency, let's compare it to C++ using a **prime number calculation benchmark**. As shown in *Fig.1*, if we take the same algorithm and run it in both languages, C++ executes the task roughly twice as fast as JavaScript. This performance gap exists because C++ is statically compiled into highly optimized machine code, whereas JavaScript relies on **runtime optimizations** to improve execution. However, what's impressive is that JavaScript, despite its dynamic nature, still achieves a **comparable level of performance**. The key question is: *How does JavaScript accomplish this?* To answer that, we need to understand the role of JavaScript engines, particularly how they use **baseline and optimizing compilers** to bridge the performance gap.

UNDER THE HOOD : JAVASCRIPT EXECUTION

We all know that V8 (the JavaScript engine used in **Chrome** and **Node.js**) is responsible for executing JavaScript code in web browsers and server environments. When JavaScript runs, it first goes through **parsing**, where it's converted into an **Abstract Syntax Tree (AST)**, which represents the structure of the code. Then, **V8's baseline compiler (Ignition)** quickly translates the AST into **bytecode** and then into machine code for immediate execution. This compilation is fast but lacks deep optimizations, ensuring the code runs as soon as possible. In the example (as shown in *Fig. 2*), **add({ x: 42 })**, **add({ x: 7 })**, and similar calls are handled by the baseline compiler, which generates simple, generic machine code. This approach prioritizes startup speed over efficiency, allowing JavaScript's dynamic nature to work smoothly.

If a function is called frequently (a **"hot" function**), the JavaScript engine recompiles it using an **optimizing compiler (TurboFan)**. This compiler analyzes past executions and makes an assumption—here, it assumes all objects passed to **add** contain an **x** property with a **number** value. Using this assumption, it generates **highly optimized machine code,** significantly improving execution speed. This process is part
balances
of **Just-In-Time (JIT) compilation**, which performance and flexibility by adapting to real-time usage patterns. As a result, optimized code runs much faster than the baseline version, making JavaScript competitive with traditionally compiled languages like C++.

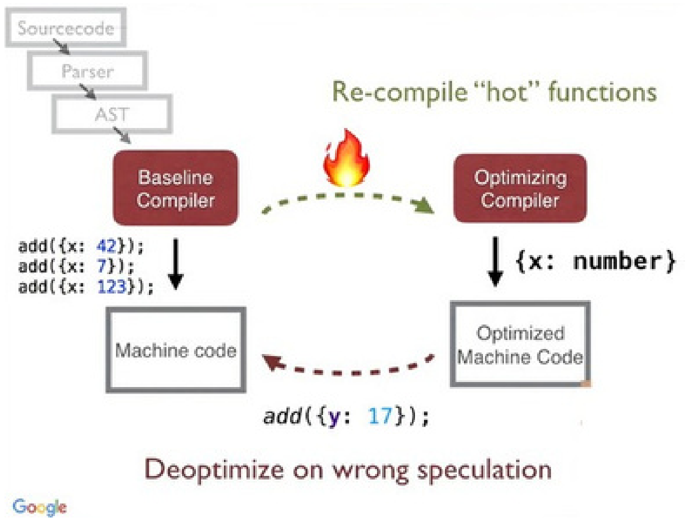

Fig.2

However, if a new function call introduces an unexpected input, such as **add({ y: 17 })**, the assumption breaks. The optimized code expects an **x** property, so when it's missing, the engine must deoptimize, reverting to the slower baseline version. This constant switching between optimized and deoptimized code, known as bailout, can cause serious performance issues. While JavaScript engines like **V8 (Chrome, Node.js)**, **SpiderMonkey (Firefox)**, and **JavaScriptCore (Safari)** are designed to handle this process efficiently, frequent deoptimizations can slow down applications. To avoid this, developers should write consistent **code patterns**, ensuring that functions receive similar types of objects to prevent unnecessary performance drops.

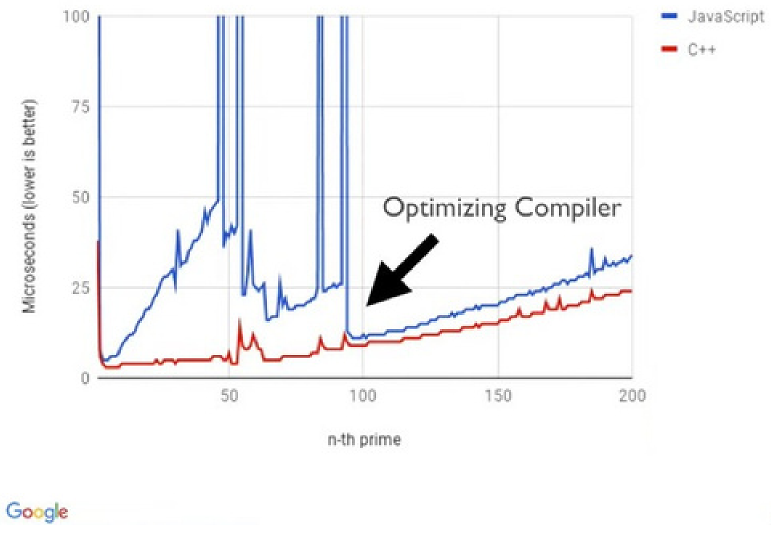

Fig.3

This graph compares (as shown in *Fig. 3*) **JavaScript (blue)** and **C++ (red)** in calculating prime numbers, measured in microseconds (lower is better). Initially, JavaScript is much slower as it runs through the **Ignition interpreter**, prioritizing quick execution over efficiency. Over time, **V8's TurboFan optimizing compiler** detects patterns, optimizes the code, and significantly boosts performance—seen where the blue line drops. However, even after optimization, JavaScript retains some overhead compared to C++, which is **statically compiled** and consistently faster for such computational tasks.

OPTIMIZED VS NOT OPTIMISED : COMPARISION

This graph (as shown in *Fig.4*) highlights the impact of **JavaScript's optimizing compiler (TurboFan)** when calculating prime numbers. In the **"Not Optimized"** graph (top), JavaScript (blue) is much slower than C++ (red) because it runs using the **Ignition interpreter**, which prioritizes quick startup over efficiency. As the input size grows, execution time increases rapidly. However, in the **"Optimized"** graph (bottom), TurboFan detects repeated patterns, recompiles the code, and significantly improves performance. This demonstrates how **Just-In-Time (JIT) compilation** helps JavaScript handle computational tasks more efficiently by converting frequently executed code into optimized machine code

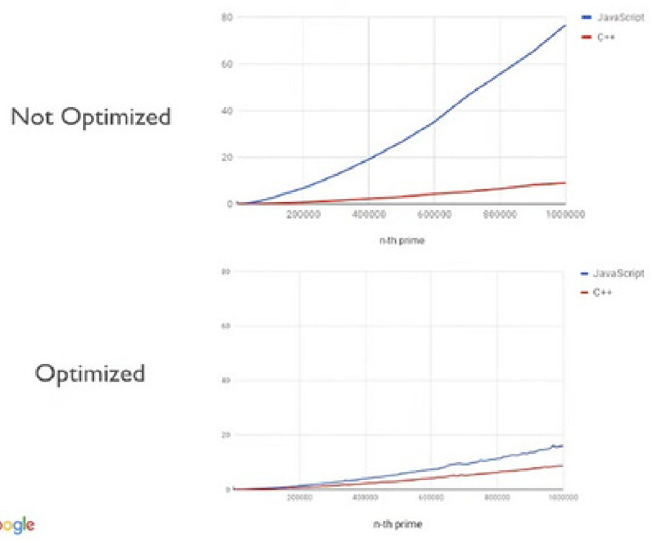

Fig.4

In conclusion, JavaScript engines like **V8 (used in Chrome and Node.js)** use **interpretation (Ignition)** for fast startup and **JIT compilation (TurboFan)** for optimization. While JavaScript is initially slower, **adaptive optimizations** help narrow the gap with C++. However, its dynamic nature can cause **deoptimizations** if code structure changes unexpectedly. To maximize performance, developers should write predictable code and avoid unnecessary type changes. Although JavaScript won't match C++ in raw speed, runtime optimizations make it powerful for **web development**, **server-side computing (Node.js)**, and even **game development**. Understanding **V8's optimizations** helps developers write **efficient**, **high-performance** code that runs smoothly across different environments.