Ефремов А.С., Потапов С.А. Основные механизмы оптимизации языка JAVASCRIPT и их влияние на производительность// Международный журнал информационных технологий и энергоэффективности.– 2026. –Т. 11 № 6(68) Ч. 1 с. 70– 78

Международный журнал информационных технологий и энергоэффективности

Сайт журнала:
http://www.openaccessscience.ru/index.php/ijcse/



УДК 004.4'41

ОСНОВНЫЕ МЕХАНИЗМЫ ОПТИМИЗАЦИИ ЯЗЫКА JAVASCRIPT И ИХ ВЛИЯНИЕ НА ПРОИЗВОДИТЕЛЬНОСТЬ

[1]Ефремов А.С., [2]Потапов С.А.
ФГАОУ ВО "САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ АЭРОКОСМИЧЕСКОГО ПРИБОРОСТРОЕНИЯ", Санкт-Петербург, Россия (190000, город Санкт-Петербург, Большая Морская ул., д.67 лит. а), e-mail: [1]andrey200ok@yandex.ru [2]potapo.potapov2014@yandex.ru

Данная работа посвящена анализу механизмов оптимизации JavaScript-кода, применяемых в системе выполнения V8. Рассматривается многоуровневая архитектура движка, включающая интерпретатор Ignition, базовый JIT-компилятор Sparkplug и оптимизирующие компиляторы Maglev и TurboFan. Особое внимание уделяется JIT-компиляции, On-Stack Replacement, сбору статистики выполнения, inline caching, hidden classes, elements kinds и внутреннему представлению малых целых чисел SMI. Показано, что эффективность оптимизации зависит от стабильности типов, форм объектов и структуры массивов. В практической части выполнено сравнение производительности JavaScript-кода при разных максимальных уровнях оптимизации V8 с использованием JetStream 3.

Ключевые слова: Javascript, v8, jit-компиляция, ignition, sparkplug, maglev, turbofan, osr, hidden classes, inline caching, динамическая оптимизация.

COMMON MECHANISMS OF JAVASCRIPT OPTIMIZATION AND THEIR IMPACT ON PERFORMANCE

[1]Efremov A.S., [2]Potapov S.A.
ST. PETERSBURG STATE UNIVERSITY OF AEROSPACE INSTRUMENTATION, St. Petersburg, Russia (190000, St. Petersburg, Bolshaya Morskaya str., 67 lit. a), e-mail: [1]andrey200ok@yandex.ru [2]potapo.potapov2014@yandex.ru

This paper analyzes JavaScript code optimization mechanisms used in the V8 execution engine. The study examines V8's multi-tier architecture, including the Ignition interpreter, the Sparkplug baseline JIT compiler, and the optimizing compilers Maglev and TurboFan. Special attention is paid to JIT compilation, On-Stack Replacement, runtime feedback collection, inline caching, hidden classes, elements kinds, and the internal representation of small integers known as SMI. The paper shows that optimization efficiency depends on type stability, object shapes, and array structure. The experimental part compares JavaScript performance at different maximum V8 optimization levels using the JetStream 3 benchmark suite.

Keywords: Javascript, v8, jit compilation, ignition, sparkplug, maglev, turbofan, osr, hidden clas-ses, inline caching, dynamic optimization.

Введение

JavaScript изначально создавался как язык сценариев для веб-страниц, однако со временем стал использоваться в сложных клиентских интерфейсах, серверных приложениях и кроссплатформенных системах. Это повысило требования к производительности программ. Простая интерпретация кода сопровождается накладными расходами на обработку инструкций, диспетчеризацию операций и работу с динамическими типами. Кроме того,

JavaScript допускает изменение структуры объектов во время выполнения и использует прототипную модель наследования, что затрудняет предварительную статическую оптимизацию [1].

Одна и та же операция в JavaScript может иметь разные варианты выполнения. Например, оператор сложения может использоваться как для арифметики, так и для конкатенации строк, а доступ к свойству объекта зависит от его формы, порядка добавления свойств и цепочки прототипов. Поэтому система выполнения не может заранее во всех случаях создать специализированный машинный код.

Одним из основных способов решения этой проблемы является динамическая оптимизация. Сначала движок выполняет код универсальным способом, затем собирает информацию о реальных типах значений, вызовах функций и структурах объектов, после чего использует эти сведения для генерации более эффективного машинного кода [1].

В данной работе рассматривается V8 — открытая система выполнения JavaScript и WebAssembly, разрабатываемая Google. V8 используется в Chrome, Node.js и может встраиваться в C++-приложения [2]. В Node.js он отвечает за выполнение JavaScript-кода вне браузера, что делает его значимым как для клиентской, так и для серверной разработки [3].

Постановка задачи

Проблема исследования заключается в том, что динамическая природа JavaScript не позволяет заранее надёжно определить типы данных, формы объектов и характер операций. Следовательно, для повышения производительности требуется использовать механизмы, способные адаптировать выполнение программы к её фактическому поведению.

Цель работы — проанализировать основные механизмы оптимизации JavaScript в движке V8 и оценить их влияние на производительность программ.

Для достижения цели необходимо решить следующие задачи: рассмотреть причины проблем производительности JavaScript; описать архитектурные уровни V8; проанализировать JIT-компиляцию и On-Stack Replacement; рассмотреть feedback vector, inline caching, SMI, elements kinds и hidden classes; оценить влияние уровней оптимизации V8 на производительность с использованием набора бенчмарков JetStream 3.

Методика исследования

Исследование включает теоретический и экспериментальный этапы. На теоретическом этапе анализируются архитектура V8 и механизмы оптимизации JavaScript-кода: Ignition, Sparkplug, Maglev, TurboFan, JIT-компиляция, OSR, feedback vector, inline caching, SMI, elements kinds и hidden classes [1, 2, 4–13].

На экспериментальном этапе сравнивается производительность JavaScript-кода при разных максимальных уровнях оптимизации V8. Для оценки используется набор бенчмарков JetStream 3, предназначенный для измерения производительности JavaScript и WebAssembly [14]. Основной метрикой является Score: чем выше его значение, тем выше производительность.

Методика эксперимента включает запуск одного и того же набора тестов при разных уровнях оптимизации, фиксацию общего рейтинга Score, сравнение результатов между

Ignition, Sparkplug, Maglev и TurboFan, а также расчёт прироста производительности между соседними уровнями и относительно базового режима Ignition.

Архитектура оптимизации V8

V8 использует многоуровневую систему выполнения, задача которой состоит в балансе между быстрым запуском программы, экономией памяти и высокой скоростью выполнения горячего кода. Вместо немедленной компиляции всего кода в максимально оптимизированный машинный код движок постепенно переводит функции между уровнями в зависимости от частоты их использования и стабильности поведения [4].

На первом уровне исходный JavaScript-код преобразуется в байткод и выполняется интерпретатором Ignition. Это обеспечивает быстрый запуск программы и позволяет не тратить ресурсы на компиляцию редко выполняемых функций [4].

Следующий уровень — Sparkplug, базовый JIT-компилятор. Он быстро преобразует байткод в машинный код без сложных оптимизаций. Такой подход снижает накладные расходы интерпретации, но сохраняет низкую стоимость компиляции [5].

Для более часто выполняемого кода применяется Maglev. Он использует данные профилирования и быстро создаёт достаточно эффективный машинный код для горячих участков программы [6].

Наиболее мощным уровнем является TurboFan. Он применяется к часто выполняемому и достаточно стабильному коду, используя данные профилирования для более сложных оптимизаций: специализации операций, встраивания функций, устранения избыточных вычислений и оптимизации циклов [4].

Таким образом, V8 реализует постепенную оптимизацию: Ignition обеспечивает запуск и сбор первичной информации, Sparkplug снижает накладные расходы интерпретации, Maglev быстро оптимизирует горячий код, а TurboFan обеспечивает максимальную производительность на наиболее стабильных участках программы.

JIT-компиляция и On-Stack Replacement

JIT-компиляция выполняется во время работы программы и позволяет учитывать её фактическое поведение: типы значений, формы объектов, целевые функции вызовов и характер операций [1]. На основе этой информации V8 создаёт специализированный машинный код.

Оптимизация в V8 является спекулятивной. Компилятор формирует предположения на основе ранее собранной статистики. Например, если операция сложения обычно выполняется над малыми целыми числами, для неё может быть создан быстрый специализированный путь. Если обращение к свойству происходит у объектов одной формы, общий поиск свойства может быть заменён доступом по известному смещению [7].

Если предположения перестают выполняться, V8 выполняет деоптимизацию: оптимизированный код заменяется более универсальным вариантом выполнения. Это позволяет сохранить корректность программы при изменении типов данных или структур объектов [7].

Для перехода между версиями кода используется On-Stack Replacement. Этот механизм позволяет заменить текущую версию исполняемого кода прямо во время работы функции. Он

особенно важен для длинных циклов, поскольку даёт возможность применить оптимизацию без ожидания повторного вызова функции.

Базовые механизмы оптимизации

Оптимизация V8 основана на совокупности механизмов, которые делают динамическое поведение JavaScript более предсказуемым для движка. К таким механизмам относятся SMI, feedback vector, elements kinds, hidden classes и inline caching.

Представление малых целых чисел (SMI)

Согласно спецификации ECMAScript, тип Number представляет числовые значения в формате double-precision 64-bit binary format IEEE 754 [8]. Однако внутри V8 для малых целых чисел используется специальное представление SMI. Оно позволяет хранить небольшие целые значения непосредственно в машинном слове без выделения отдельного объекта в куче. Это снижает накладные расходы и ускоряет арифметические операции [9].

Если значение выходит за пределы диапазона SMI или требует представления с плавающей точкой, V8 переходит к более общему способу хранения. Поэтому стабильное использование малых целых чисел в горячем коде повышает вероятность применения быстрых путей выполнения.

Feedback vector

Feedback vector используется для накопления информации о поведении функции во время выполнения. В нём сохраняются сведения об обращениях к свойствам, вызовах функций, арифметических операциях, сравнениях и доступе к элементам массивов [7].

Например, для операции сложения может быть зафиксировано, что она чаще выполняется над целыми числами, а для обращения к свойству — что объект имеет одну и ту же форму. Эти данные используются inline caching и оптимизирующими компиляторами. Если поведение функции стабильно, V8 может создать специализированный машинный код. Если поведение меняется, feedback обновляется, а оптимизированный код может быть деоптимизирован.

Elements kinds

Elements kinds — механизм классификации массивов в V8. Он нужен, поскольку массивы JavaScript могут содержать значения разных типов и изменять структуру во время выполнения. V8 различает массивы малых целых чисел, массивы чисел с плавающей точкой, массивы обычных элементов, а также плотные массивы и массивы с пропусками [10].

Плотные однотипные массивы обрабатываются быстрее, так как требуют меньше проверок. Переходы между elements kinds обычно направлены от более специализированных представлений к более общим. Например, добавление вещественного числа в массив SMI переводит его к double-представлению, а добавление строки или объекта — к ещё более универсальному виду [10]. Поэтому смешивание типов и появление пропусков снижает эффективность оптимизации.

Hidden classes

Hidden classes используются для ускорения доступа к свойствам объектов. JavaScript позволяет изменять структуру объектов во время выполнения, поэтому V8 внутренне описывает форму объекта с помощью структур Map. Hidden class хранит набор свойств, порядок их добавления и расположение значений в памяти [11].

Если несколько объектов создаются одинаково и получают свойства в одном порядке, они могут иметь одну hidden class. В этом случае доступ к свойству выполняется не как полный поиск по имени, а как обращение по известному смещению [11].

Для описания свойств используются DescriptorArray, а переходы между формами объектов фиксируются с помощью TransitionArray [12]. Важным фактором является порядок добавления свойств: объекты с одинаковым итоговым набором свойств могут иметь разные hidden classes, если свойства добавлялись в разной последовательности. Поэтому стабильное создание объектов одного типа улучшает условия для оптимизации.

Inline caching

Inline caching ускоряет доступ к свойствам и вызовы методов за счёт повторного использования информации, полученной при предыдущих выполнениях той же операции. При первом обращении к свойству V8 использует общий механизм поиска, затем сохраняет форму объекта и способ доступа к значению. При последующих обращениях движок проверяет форму объекта и, если она совпадает, использует быстрый путь [13].

Inline cache может быть мономорфным, полиморфным или мегаморфным. Мономорфный вариант наиболее эффективен, поскольку одно место программы работает с объектами одной формы. При большем числе форм эффективность специализации снижается [13].

Inline caching связан с feedback vector и hidden classes. Hidden classes описывают формы объектов, feedback vector сохраняет сведения о выполнении, а inline cache использует эти данные для ускорения конкретных операций доступа [7, 13].

Тестирование производительности при разных уровнях оптимизации

В целях эмпирической оценки влияния механизмов оптимизации на производительность была проведена серия тестов на бенчмарке JetStream 3. В качестве метрики для каждого отдельного теста использовался Score – безразмерная метрика, высчитываемся на основе среднего геометрического значения времени выполнения программы [15].

Большее значение Score соответствует лучшей производительности. Результаты данного тестирования представлены в Таблице 1.

Таблица 1 - Результаты тестирования (значение Score) с разными уровнями оптимизации на бенчмарке JetStream 3

Название теста	Ignition	Sparkplug	Maglev	Turbofan
WSL	2.01	2.66	5.45	5.61
web-ssr	151.72	185.2	236.97	234.06
validatorjs	344.09	391.16	461.6	459.65
UniPoker	105.09	115.83	692.34	815.47
typescript-lib	1.23	2.37	6.69	6.91
threejs	15.78	19.66	79.59	77.33
sync-fs	77.62	109.35	437.47	586.32

Sunspider	146.46	162.97	762.78	945.81
stanford-crypto-sha256	48.88	52.79	366.18	1030.3
stanford-crypto-pbkdf2	52.26	56.33	282.14	983.13
stanford-crypto-aes	41.44	47.82	295.91	570.62
splay	97.88	129.1	346.7	365.03
source-map-wtb	107.04	132.12	424.14	393.11
richards	63.69	75.37	1230.43	1381.64
Пregexp-octane	331.56	401.35	753.95	760.79
raytrace-public-class-fields	39.56	50.74	389.18	790.8
raytrace-private-class-fields	28.78	36.9	312.76	665.88
raytrace	50.38	59.53	542.78	755.95
proxy-vue	431.38	568.21	857.72	883.83
proxy-mobx	815.29	1043.29	1761.02	1740.01
prismjs-startup-es6	383.6	420.75	422.05	395.61
prettier-wtb	23.44	36.06	99.11	101.1
postcss-wtb	16.71	21.53	43.83	47.25
pdfjs	74.68	94.73	311.88	327.03
OfflineAssembler	99.87	126.54	298.88	284.85
octane-code-load	1345.12	1285.94	1342.87	1296.56
navier-stokes	100.24	100.53	505.92	594.61
multi-inspector-code-load	448.8	439.77	509.97	491.9
mobx-startup	65.42	131.06	233.74	232.97
ML	9.95	13.38	100.08	168.97
mandreel	5.47	6.21	94.73	157.56
lazy-collections	80.69	154.76	315.01	336.22
json-stringify-inspector	412.81	473.79	515.94	510.9
json-parse-inspector	286.73	337.6	331.9	334.44
jsdom-d3-startup	9.55	13.36	20.52	20.32
js-tokens	126.27	188.72	383.61	381.14
hash-map	44.25	55.66	623.84	896.38
gbemu	24.67	21.07	173.63	204.18
gaussian-blur	21.14	23.3	204.4	470.02
FlightPlanner	465.52	578.19	968.03	1025.15
first-inspector-code-load	287.8	284.92	269.8	272.23
esprima-next-wtb	13.7	17.29	117.81	129.82
espree-wtb	11.32	15.63	86.12	105.49

earley-boyer	163.82	208.09	1091.44	1201.03
doxbee-promise	397.46	488.61	814.36	772.23
doxbee-async	418.09	663.65	1110.83	1496.76
delta-blue	96.77	115.95	1670.57	2462.9
crypto	89.83	104.21	802.51	1773.85
chai-wtb	135.89	173.23	266.04	288.1
cdjs	41.37	49.19	307.57	341.59
Box2D	72.72	80.64	597.11	649.37
bigint-noble-ed25519	62.08	80.44	139.23	150.8
Basic	268.48	341.65	1212.89	1324.8
babylonjs-startup-es6	28.37	26.63	27.51	28.62
babylonjs-scene-es6	37.79	45.58	74.53	78.57
babylon-wtb	12.8	17.63	100.54	112.25
Babylon	204.5	259.34	1093.51	1259.21
babel-wtb	58.97	80.03	158.95	162.39
babel-minify-wtb	35.15	49.47	90.47	128.41
async-fs	130.61	190.6	388.17	506.76
Air	303.29	337.08	726.13	768.94
ai-astar	141.96	177.31	884.75	1026.06
acorn-wtb	8.31	11.04	69.72	76.73
Общий рейтинг	71.52	89.12	286.39	344.85

Таблица 2. - Прирост к производительности от каждого уровня оптимизации

Уровень оптимизации	Рейтинг	Прирост к предыдущему уровню (%)	Прирост относительно производительности без оптимизации (%)
Ignition	71.52		
Sparkplug	89.12	24.61%	24.61%
Maglev	286.39	221.35%	300.43%
TurboFan	344.85	20.41%	382.17%

Переход от Ignition к Sparkplug увеличил производительность на 24.61%. Наиболее значимый прирост был получен при переходе от Sparkplug к Maglev: рейтинг вырос с 89.12 до 286.39, что соответствует увеличению на 221.35%. Переход от Maglev к TurboFan дал дополнительный прирост 20.41%. Итоговое ускорение TurboFan относительно Ignition составило 382.17%.

Результаты показывают, что наибольший вклад в повышение производительности даёт оптимизирующая JIT-компиляция, использующая статистику выполнения. При этом максимальный уровень оптимизации не всегда является лучшим для каждого отдельного теста, поскольку результат зависит от структуры кода, стабильности типов и стоимости самой оптимизации.

Заключение

В ходе работы были рассмотрены основные механизмы оптимизации JavaScript, применяемые в движке V8. Показано, что производительность обеспечивается совокупностью решений: многоуровневой JIT-компиляцией, сбором статистики выполнения, inline caching, hidden classes, elements kinds, SMI, спекулятивной оптимизацией и деоптимизацией.

V8 сначала выполняет код универсальным способом, а затем постепенно создаёт специализированные версии машинного кода для часто исполняемых участков. Такой подход позволяет учитывать динамическую природу JavaScript и снижать накладные расходы, связанные с изменяемыми типами и структурами объектов.

Практическое тестирование подтвердило влияние уровней оптимизации на производительность. Общий рейтинг JetStream 3 вырос с 71.52 в режиме Ignition до 344.85 в режиме TurboFan, что соответствует приросту 382.17%. Наибольший вклад внёс Maglev, обеспечивший резкий рост производительности после базовой компиляции Sparkplug. Это подтверждает, что профилирующая JIT-компиляция является одним из ключевых механизмов ускорения современных JavaScript-приложений.

Список литературы

1. Жуйков Р., Мельник Д., Бучацкий Р., Варданян В., Иванишин В., Шарыгин Е. Методы динамической и предварительной оптимизации программ на языке JavaScript // Труды ИСП РАН. – Т. 26. – №1. – 2014. – С. 297-314. https://doi.org/10.15514/ISPRAS-2014-26(1)-10
2. Официальный сайт V8. What is V8? [Электронный ресурс]. – Режим доступа: https://v8.dev (дата обращения: 25.04.2026)
3. Node.js Documentation. The V8 JavaScript Engine [Электронный ресурс]. – Режим доступа: https://nodejs.org/en/learn/getting-started/the-v8-javascript-engine (дата обращения: 25.04.2026)
4. Официальный сайт V8. Launching Ignition and TurboFan [Электронный ресурс]. – Режим доступа: https://v8.dev/blog/launching-ignition-and-turbofan (дата обращения: 26.04.2026)
5. Официальный сайт V8. Sparkplug – a non-optimizing JavaScript compiler [Электронный ресурс]. – Режим доступа: https://v8.dev/blog/sparkplug (дата обращения: 26.04.2026)
6. Официальный сайт V8. Maglev – V8s Fastest Optimizing JIT [Электронный ресурс]. – Режим доступа: https://v8.dev/blog/maglev (дата обращения: 26.04.2026)
7. Варданян В. Г. Методы оптимизации программ на языке JavaScript, основанные на статистике выполнения программы // Труды ИСП РАН. – 2016. – Т. 28. – №1. – С. 520. https://doi.org/10.15514/ISPRAS-2016-28(1)-1
8. ECMA International. ECMAScript 2023 Language Specification. ECMA-262 [Электронный ресурс]. – Режим доступа: https://262.ecma-international.org/14.0/ (дата обращения: 27.04.2026)
9. Официальный сайт V8. Pointer Compression in V8 [Электронный ресурс]. – Режим доступа: https://v8.dev/blog/pointer-compression (дата обращения: 27.04.2026)
10. Официальный сайт V8. Elements kinds in V8 [Электронный ресурс]. – Режим доступа: https://v8.dev/blog/elements-kinds (дата обращения: 27.04.2026)
11. Официальный сайт V8. Maps (Hidden Classes) in V8 [Электронный ресурс]. – Режим доступа: https://v8.dev/docs/hidden-classes (дата обращения: 28.04.2026)

12. Официальный сайт V8. Fast properties in V8 [Электронный ресурс]. – Режим доступа: https://v8.dev/blog/fast-properties (дата обращения: 28.04.2026)
13. Bynens M. JavaScript engine fundamentals: Shapes and Inline Caches [Электронный ресурс]. – Режим доступа: https://mathiasbynens.be/notes/shapes-ics (дата обращения: 28.04.2026)
14. WebKit. Introducing the JetStream 3 Benchmark Suite [Электронный ресурс]. – Режим доступа: https://webkit.org/blog/17899/introducing-the-jetstream-3-benchmark-suite (дата обращения: 29.04.2026).

References

1. R. Zhuikov, D. Melnik, R. Buchatsky, V. Vardanyan, V. Ivanishin, E. Sharygin. Methods of dynamic and preliminary optimization of JavaScript programs. Proceedings of ISP RAS. - Vol. 26. - No. 1. - 2014. - pp. 297-314. https://doi.org/10.15514/ISPRAS-2014-26(1)-10
2. Official V8 website. What is V8? [Electronic resource]. - Access mode: https://v8.dev (date of access: 04/25/2026)
3. Node.js Documentation. The V8 JavaScript Engine [Electronic resource]. – Access mode: https://nodejs.org/en/learn/getting-started/the-v8-javascript-engine (date accessed: 25.04.2026)
4. Official V8 website. Launching Ignition and TurboFan [Electronic resource]. – Access mode: https://v8.dev/blog/launching-ignition-and-turbofan (date accessed: 26.04.2026)
5. Official V8 website. Sparkplug – a non-optimizing JavaScript compiler [Electronic resource]. – Access mode: https://v8.dev/blog/sparkplug (date accessed: 26.04.2026)
6. Official V8 website. Maglev – V8s Fastest Optimizing JIT [Electronic resource]. – Access mode: https://v8.dev/blog/maglev (date accessed: 26.04.2026)
7. Vardanyan V. G. Methods for optimizing JavaScript programs based on program execution statistics // Proceedings of ISP RAS. – 2016. – Vol. 28. – No. 1. – p. 520. https://doi.org/10.15514/ISPRAS-2016-28(1)-1
8. ECMA International. ECMAScript 2023 Language Specification. ECMA-262 [Electronic resource]. – Access mode: https://262.ecma-international.org/14.0/ (date accessed: 27.04.2026)
9. Official website of V8. Pointer Compression in V8 [Electronic resource]. – Access mode: https://v8.dev/blog/pointer-compression (accessed date: 27.04.2026)
10. Official V8 website. Elements kinds in V8 [Electronic resource]. – Access mode: https://v8.dev/blog/elements-kinds (accessed date: 27.04.2026)
11. Official V8 website. Maps (Hidden Classes) in V8 [Electronic resource]. – Access mode: https://v8.dev/docs/hidden-classes (accessed date: 28.04.2026)
12. Official V8 website. Fast properties in V8 [Electronic resource]. – Access mode: https://v8.dev/blog/fast-properties (accessed date: 28.04.2026)
13. Bynens M. JavaScript engine fundamentals: Shapes and Inline Caches [Electronic resource]. – Available at: https://mathiasbynens.be/notes/shapes-ics (Accessed: 28.04.2026)
14. WebKit. Introducing the JetStream 3 Benchmark Suite [Electronic resource]. – Available at: https://webkit.org/blog/17899/introducing-the-jetstream-3-benchmark-suite (Accessed: 29.04.2026).