SUMMARY

Dette kandidatspeciale fortsætter det eksisterende arbejde med benchmarking af JavaScript engines. Eksisterende arbejde har haft fokus på at benchmarke algoritmer eller hele projekter, mens at denne kandidatspeciale fokuserer på at benchmarke små mønstre, som kun har en enkelt forskel mellem dem. Målet med dette er at bestemme, hvilke mønstre der opnår det samme resultat bruger mindst energi. Som intuition til hvorfor dette er forskelligt for engines som implementerer den funktionelt samme standard, er generel engine-struktur forklaret som et baggrundskapitel. De små mønstre, som bruger mere energi, kan blive skrevet som regler til statisk analyse. Det vil sige at man kan finde potentielle optimeringer, uden at eksekvere kildekoden.

Der inkluderes 7 JavaScript engines, inklusive dem som bruges til de mest populære browsere og runtimes. Disse 7 engines kører 25 kategorier af benchmarks med i alt 68 forskellige mønstre. Hver kombination af engine og mønstre er eksekveret 10 gange, for hvert antal af iterationer. Det vil sige at både vækst, median, og variabilitet er dokumenteret. Antallet af iterationer bliver automatisk bestemt for hver kombination af engine og kategori, ved at fordoble antallet af iterationer indtil at eksekveringen tager over 10 sekunder. Dette skyldes, at de forskellige engines og kategorier har meget forskellige ydeevner. Det primære mål er ikke at sammenligne engines med hinanden, men kun at bestemme hvilke små mønstre der bruger mindst energi for hver engine.

Denne kandidatspeciale bidrager med benchmarking og evaluering af energiforbruget af små mønstrer som opnår det samme resultat under eksekvering. Målinger er foretaget ved brug af Running Average Power Limit, som måler CPU'ens energiforbrug. Et bash script kører hvert mønster med hver engine og logger energiforbruget, hvor lang tid det tog, og det højeste hukommelsesforbrug.

Resultaterne viser at de forskellige engines har store forskelle i deres ydeevner, baseret på hvilket mønstre de eksekverer. Nogle mønstrer bruger mindst energi på nogle engines, mens at de samme mønstrer ikke bruger mindst energi på andre engines i forhold til andre mønstrer. Dette indikerer, at regler til statisk analyse ikke kan baseres på målinger foretaget på en enkelt engine og generaliseres til en global regel. Nogle engines viser en højere median relativ energiforbrug end andre. Yderligere viser nogle engines højere variabilitet. Dette betyder, at for nogle engines er det vigtigere, at man tænker sig om end for andre engines. Derfor betyder valget af engine noget for energiforbruget, men det er ikke forudsigeligt eller så let som at kode A er bedre end

kode B. Resultaterne er baseret på kontrollerede benchmarks, som ikke nødvendigvis afspejler den virkelige verden.

Mønstre kan skrives som regler til statisk analyse, som kan informere udvikleren om, hvad de kan forvente af et mønster over et andet, afhængigt af hvilken engine de bruger. Nogle regler er simple og skal bare erstatte hvilken metode der kaldes, mens andre er mere involverede og kræver kontekst. Derfor er det ikke trivielt at automatisk erstatte kode med et mere energivenligt alternativ.

Samlet set konkluderes det, at energiforbruget af JavaScript engines er uforudsigeligt, da det afhænger af engine og kontekst, men målbart. Derfor skal regler for statisk analyse ikke generaliseres baseret på en enkelt engine.

Energy-Aware Static Analysis for JavaScript: Suggesting Efficient Code Patterns

P10 Project Report

cs-26-pt-10-02

Aalborg University

Software



AALBORG UNIVERSITY

STUDENT REPORT

Fourth Candidate Semester w/ Department of Computer Science
Software, Master of Science (MSC) in Engineering
Selma Lagerløfs vej 300
9220, Aalborg
cs.aau.dk

Title:
Energy-Aware Static Analysis for JavaScript: Suggesting Efficient Code Patterns

Theme:
Programming Technology

Project Period:
Spring 2026 - 4th Candidate Semester
(10th Semester)

Project Group:
cs-26-pt-10-02

Participants:
Anders Nyby Christensen

Supervisor:
Bent Thomsen
Søren Kejser Jensen

Pages: 61

Date of Completion:
June 8, 2026

The content of this report is freely available, but publication (with reference) may only be pursued due to agreement with the author.

Abstract:

This thesis contributes by measuring and evaluating the energy efficiency of 68 JavaScript code patterns across 7 JavaScript engines. Prior work has primarily focused on algorithms or entire applications, whereas this study isolates small variations in code that achieve the same output. The motivation here is to explore the impact of small differences and whether they can be written as rules for static analysis.
The benchmarks are executed multiple times under controlled conditions, with increasing numbers of iterations, to document growth. Energy consumption, time, and peak memory are measured and logged. The experiment is designed with iteration numbers specific to each engine. The results show that energy efficiency varies significantly across engines and patterns. A pattern that is more efficient on one engine can be inefficient in another. Some patterns are always more efficient. Therefore, rules must be based on per-engine measurements rather than generalizations. Finally, it is demonstrated that a pattern can be expressed as a rule that enables the automatic detection of potentially inefficient code patterns without executing the code.

CONTENTS

1 Preface 4

2 Introduction 5
- 2-A Motivation: Energy Efficiency in Software . . . 5
- 2-B JavaScript as a Relevant Platform . . . 6
- 2-C Problem: Variability in Execution Efficiency . . . 6
- 2-D Aim and Contribution of This Thesis . . . 6
- 2-E Thesis Structure . . . 7

3 Research Gap 7
- 3-A Prior Work on Language-Level Energy Efficiency . . . 8
- 3-B Evidence of Engine-Level Variability . . . 8
- 3-C Limitation: Lack of Fine-Grained Analysis . . . 9
- 3-D Research Gap Summary . . . 10
- 3-E Research Questions . . . 10

4 Background 11
- 4-A JavaScript Engine Architecture . . . 11
- 4-B Sources of Efficiency Variability in JavaScript . . . 13
- 4-C Energy Measurement Using RAPL . . . 14
- 4-D Benchmarking and Measurement Challenges . . . 15
- 4-E Static Analysis and Linting . . . 16
- 4-F JavaScript Performance Guidelines in Practice . . . 17

5 Methodology 18
- 5-A JavaScript Engine Setup . . . 18
- 5-B Benchmark Selection . . . 18
- 5-C Hardware and System Configuration . . . 20
- 5-D Metrics . . . 21
- 5-E Execution Procedure . . . 21

6 Results 23
- 6-A Dataset Overview . . . 23

6-B Overall Engine Differences 23
6-C Distribution of Relative Energy Consumption 27
6-D Medians and Variability 29
6-E In-depth Look at V8 30

7 Discussion 35
7-A RQ1: Choice of Engine 35
7-B RQ2: Engine-Dependent Optimizations 35
7-C RQ3: Static Analysis Rules 37
7-D Limitations 37

8 ESLint Rules 38
8-A Defining a Rule 38

9 Conclusion 41
9-A Future Work 42

Glossary 44

Appendix 50

A Engine Worst-Cases 50

B ESLint Rules 57

1. PREFACE

This master's thesis is authored by a single student at Aalborg University from 02-02-2026 to 08-06-2026 for the 4th Candidate Semester.

AI-based tools, primarily ChatGPT, have been used to aid decision-making by brainstorming pros and cons and by generating code early in the process to quickly explore data, but have not been used to generate any text or code for the work presented in this thesis. Grammarly has been used to quickly find and fix grammatical errors, primarily pesky commas.

The code is available on GitHub [5] along with steps for reproducibility and pre-computed figures.

The author expresses gratitude to his supervisors, Bent Thomsen and Søren Kejser Jensen.

2. INTRODUCTION

Previous work has examined the energy efficiency of programming languages [27, 45] and the engines that execute programs written in those languages [4, 53]. However, relatively little effort has been devoted to translating measurements into practical guidance for developers to write sustainable code. This project focuses on JavaScript, the programming language of the web, and derives per-engine rules for optimizing it.

A. Motivation: Energy Efficiency in Software

In computer science, energy efficiency is a combination of efficient hardware and efficient code. The Green Software Foundation offers standards for both hardware and software [14], with current efforts focusing on AI. This project focuses solely on code, as it is more readily modifiable and has strong potential for efficiency improvements at scale without physically replacing manufactured computer parts. The Green Software Foundation denotes a Software Carbon Intensity (SCI) score for measuring carbon emissions of software [13]. They provide the formula $SCI = (E * I + M)$ per R. E is the energy consumed, I is the carbon intensity of the energy that is used, M is the emissions from manufacturing the hardware, and R is the unit of work. This project does not account for the carbon intensity of energy sources or hardware manufacturing, as the energy source cannot be accounted for, and optimizing the software does not change manufacturing. Rather, it considers only the energy required per unit of work, i.e., E per R. Furthermore, software developers do not necessarily control the hardware.

The World Bank estimates that the Information and Communication Technology (ICT) sector consumed 1,183 TWh in 2022 [54], illustrating the scale to which small improvements can add up globally for the ICT sector. In comparison, Denmark consumed 35.5 TWh in 2022 across all sectors. This means that the ICT sector alone consumes 33 times as much energy as Denmark, a modern country with a population of 6 million [12].

Minimizing energy costs is relevant for a multitude of reasons. It is in the interests of both software and hardware providers and consumers to extend battery life, reduce energy costs, and ensure a sustainable future for generations to come. Therefore, energy efficiency is a real engineering concern.

B. JavaScript as a Relevant Platform

Software runs on everything from phones and smartwatches to washing machines and cloud servers. JavaScript is chosen because of its widespread use across backend servers and consumer devices [25, 43]. JavaScript is a language that runs in both web browsers and server-side environments via runtimes such as Node.js [32]. This large-scale use implies that guiding developers to optimize small patterns, rather than just rethinking algorithms, can yield substantial global benefits. Small code patterns are fragments of code, such as loop structures or method calls, that can be changed independently of the overall algorithm they are part of. Finding the most efficient code patterns at this level can yield benefits across all algorithms that use small fragments with multiple implementations. One example is determining whether a number is odd or even using bitwise operations versus modulo.

C. Problem: Variability in Execution Efficiency

The energy use of a program can vary drastically even when performing the same task, as shown in previous work by Pereira et al. [45], which compared different programming languages for the same problem. As the most extreme example, in the *Ackermann* benchmark, the Rust language consumed 41.69 J, whereas Perl consumed 14,475.82 J. As shown in my previous work [4], when keeping the programming language the same and varying only the engines that implement it, there are still substantial differences in energy efficiency. Engine versions, engine implementations, and program implementations all significantly contribute to energy efficiency, as described in Section 3-B. This is further supported by the work of Selakovic and Pradel, who observed performance fluctuations across versions of the V8 engine [53]. The work of Kumar et al. supports this idea by evaluating functionally equivalent patterns in the Java language [30]. Therefore, it cannot be expected that developers can reliably reason, or rather guess, about energy use.

D. Aim and Contribution of This Thesis

This work aims to evaluate micro-optimization patterns in JavaScript across engines and assess their impact by measuring energy consumption. The goal is not to determine which engine is best, but rather to identify which code patterns yield higher efficiency and which have no impact or an adverse effect. This data is then used to derive rules for static analysis that can inform developers about said patterns. Static analysis is chosen over other methods, such as runtime profiling, for its immediate developer feedback without executing the code in question and its

potential for automatic fixes through IDE integration. Furthermore, if this idea is successful, developers can avoid the expensive overhead of constantly running their code.

E. Thesis Structure

This section outlines the thesis structure and describes the purpose of each chapter.

3. Research Gap - This chapter reviews existing work in JavaScript energy efficiency to identify limitations and behavior of engines, for the motivation of why this work is necessary. Finally, research questions are proposed.

4. Background - This chapter introduces all the fundamental concepts that provide an intuition for how JavaScript engines can behave so differently, including general JavaScript engine architecture, compilation strategies, and what tools are used for benchmarking. A brief overview of static analysis is also provided.

5. Methodology - This chapter explains how the experiment is designed. This includes how the benchmark environment is configured, how the measurement tools are used, and how reproducibility is ensured.

6. Results - This chapter presents the results, primarily through figures, and provides insights by highlighting what the data shows.

7. Discussion - This chapter analyses, interprets, and discusses the raw data. This chapter also discusses the feasibility of rules for static analysis.

8. ESLint Rules - This chapter describes how to implement a static analysis rules in ESLint.

9. Conclusion - This chapter summarizes the purpose and contributions of this thesis, discusses the limitations, and guides future research in both JavaScript energy efficiency research and static analysis tools. This chapter also lists future work, i.e., ways that this research could continue or be done differently.

3. Research Gap

Prior work has focused on comparing the energy efficiency of programming languages [27, 45] and comparing the energy efficiency of multiple implementations of the same programming language, i.e., JavaScript engines [4, 53]. It shows that engines behave differently, and the next step is to determine which patterns are more efficient on which engines.

A. Prior Work on Language-Level Energy Efficiency

The work of Pereira et al. [45] and van Kampen et al. [27] suggests that, while there are differences in energy consumption across programming languages that accomplish the same task, the language has no impact on energy beyond runtime. Pereira et al. [45] show that when implementing algorithms, e.g., Sieve of Eratosthenes for finding prime numbers, the languages Pascal and C++ consume 0.02 J and 0.03 J, respectively, whereas the language Chapel consumes 2,280.27 J. The runtime ranges from 3 milliseconds to 174549 milliseconds. This illustrates that the energy consumption varies by several orders of magnitude. The Chapel language website describes the language as ”generating efficient machine code that meets or beats the performance of other languages” [48]. This highlights that a developer should make decisions based on benchmarked data rather than on what a language claims.

B. Evidence of Engine-Level Variability

Van Kampel et al. [27] showed that two different implementations of the same language can differ in performance; for example, LuaJIT (Lua with Just-in-Time (JIT)-compilation) is roughly 5 times more efficient than Lua.

Previous work comparing JavaScript performance to WebAssembly exists [7, 19, 47], but WebAssembly is not a focus of this thesis, as it is complementary to JavaScript rather than a replacement. The primary advantage of WebAssembly is that it serves as a compilation target, allowing code written in languages other than JavaScript to execute in the same web browser environment as JavaScript. WebAssembly can be up to 30% more efficient than JavaScript on mobile browsers, according to Pockstaller et al. [47].

Other work inadvertently compares JavaScript engines by evaluating the efficiency of web browsers and runtimes [8, 37], both of which are powered by them. Their inadvertent results align with previous work [4], which shows that one JavaScript engine can be up to 64 times more efficient than another when executing the same JavaScript file. While JavaScript engines are often embedded in larger environments, such as browsers or runtimes, rather than running in isolation, this thesis focuses on engines in isolation to reduce confounding or introducing factors from these environments. This allows for more direct comparisons between engine behavior and energy differences.

In other words, it is well established that different languages, and different implementations of the same language, consume vastly different amounts of energy to perform the same task.

Therefore, it has already been demonstrated that languages, engines, and algorithms vary widely in energy consumption, but the energy use differences of smaller code patterns in JavaScript remain unknown, as far as I can tell.

File	GJS	H	JSC	QJS	SM	V8	XS
binary-trees-7.js	0.063	0.169	0.015	0.415	0.038	0.017	0.775
fannkuch-redux.js	0.092	1.147	0.054	3.334	0.105	0.051	3.793
fasta-8.js	0.285	1.243	0.149	3.427	0.243	0.177	3.379
fasta.js	0.500	1.517	0.135	8.381	0.221	0.163	5.203
mandelbrot-2.js	0.077	0.454	0.067	2.699	0.070	0.068	3.648
mandelbrot-8.js	0.079	0.482	0.067	2.534	0.070	0.068	4.166
mandelbrot.js	0.077	0.479	0.067	2.730	0.071	0.069	3.777
spectral-norm-8.js	0.231	2.650	0.145	6.970	0.501	0.184	7.323
spectral-norm.js	0.129	1.958	0.111	5.862	0.398	0.129	7.150
Total	1.534	10.098	0.809	36.352	1.718	0.927	39.215

TABLE I: Total energy (J) overview of benchmarked JavaScript engines in the work of Christensen [4].

Table I is a reproduced snippet from my previous work that shows the total energy (in joules) of 7 JavaScript engines across 9 benchmarks [4]. Each row represents a benchmarked JavaScript file, and each column represents a JavaScript engine. This data shows the current state of JavaScript engine energy efficiency: there are vast differences between engines. JavaScriptCore consumes 0.135 J and QuickJS consumes 8.381 J on the *fasta.js* benchmark. Furthermore, comparing the same engines to the other *fasta* implementation, *fasta-8.js*, it was shown that JavaScriptCore increases from 0.135 J to 0.149 J (10% increase), while QuickJS decreases from 8.381 J to 3.427 J (60% decrease).

This large and inconsistent variance cannot be explained by current research, indicating that there must be patterns that are more efficient on one engine but less efficient on another.

C. Limitation: Lack of Fine-Grained Analysis

These benchmarks, however, are based on complete algorithm implementations from the Computer Language Benchmark Game (CLBG) project. Therefore, it is non-trivial to determine exactly where the impact comes from. This thesis focuses on much smaller code snippets, with the difference between snippets serving as the sole source of variation.

As shown in previous work, while different engines (or WebAssembly) can be more efficient than others, executing multiple implementations of the same algorithm can significantly affect energy efficiency. The goal is not to require that all code be written in a single language or executed with a single engine, but rather to document the patterns that affect each engine. Therefore, it is important to identify small optimization patterns that affect energy efficiency.

D. Research Gap Summary

There is already extensive evidence of differences in energy efficiency as outlined in Section 3-A and Section 3-B, with results indicating that energy is directly related to runtime as shown by van Kampen et al. [27]. Evidence also shows that different engines implementing a language, and their versions, have different impacts on energy efficiency. As far as I am aware, there is currently no fine-grained analysis of small code patterns that maps patterns to impact across engines, which is why this thesis aims to fill that gap. This is required to support developers' decision-making, and the data from the current research does not translate into actionable rules for static analysis across engines.

E. Research Questions

RQ1: "How does the choice of JavaScript engine affect the energy consumption of small JavaScript code snippets?"
The purpose of this research question is to determine whether engine choice significantly influences energy consumption, based on the code pattern. By focusing on smaller code snippets with a single source of difference, the impact on energy use is directly isolated to the variance in the patterns that achieve the same result.

RQ2: "Can a different optimization pattern that reduces energy consumption on one JavaScript engine have no effect or an adverse effect on another engine?"
The purpose of this research question is to determine whether a single optimization can be generalized across all engines without benchmarking on all engines. The work of Couto et al. finds that for mobile applications written in Java, when refactoring individually, the gains consistently exceed the losses [6]. However, when they combine multiple individual refactors, certain combinations of patterns increase energy consumption.

RQ3: "Can identified optimization patterns be expressed as static analysis rules for automated detection?"

The purpose of this research question is to explore the possibility of automatically finding antipatterns and applying the most efficient pattern for the engine in question.

This motivates an experiment that isolates smaller code patterns rather than full algorithms and evaluates them under controlled conditions across multiple JavaScript engines to determine the most efficient pattern.

4. Background

This thesis does not aim to definitively explain why every engine behaves as it does across all optimization patterns. Instead, this chapter describes how JavaScript engines work at a high level, providing intuition for why small pattern changes can lead to differences in energy efficiency across engines, for better or worse.

A. JavaScript Engine Architecture

In 1996, Internet Explorer and Netscape had their own 'JavaScript' specifications, which meant developers had to target one or the other or write more code [22]. Because of this, Netscape submitted its specification to ECMA International for standardization in 1997. Since then, JavaScript is any implementation of the JavaScript ECMA specification [24]. This has led to many implementations, but they are all expected to produce the same observable behavior for the same input code, even if implemented in different ways using various strategies.

Engines differ in many ways, including parsing strategies, compilation strategies, garbage collection, optimization heuristics, and more [11, 21, 56].

Most modern engines, such as V8, JavaScriptCore, and SpiderMonkey, produce a Abstract Syntax Tree (AST) through eager and/or lazy parsing of the source code [55, 20, 39]. This AST can then be translated into bytecode or another intermediate representation. An engine may directly translate it into more efficient machine code. Engines such as QuickJS instead generate bytecode directly without any IR, such as a syntax tree [2]. Eager parsing processes the entire source code into an AST immediately, whereas lazy parsing only processes the parts that are necessary to fulfill the current purpose, and defers the parsing of the rest until required. Alternatively, an engine such as Hermes applies a strategy called Ahead-of-Time (AOT) compilation [10], in which the source code is translated into executable code during the build step before program execution [57]. This illustrates that there are structural differences between engines, with trade-offs.

Some engines utilize JIT compilation. When executing the bytecode, a JIT-compiled engine can collect profiling data, including types and hot functions [34]. Combining the bytecode with the profiling data allows it to make educated assumptions (speculative optimization) and generate more optimized machine code. A single engine can have multiple JIT tiers as described below. The lowest tier is basic and intended for code that does not run often, as there is no need to incur the cost of optimizing code that will not run again; the benefit must outweigh the cost, and that is the battle the engines fight. When code is deemed *hot*, i.e., it runs often, the engine might tier up, i.e., be promoted to a more aggressive compiler that uses profiling data to optimize further based on assumptions. Some JIT engines have more tiers than others. If assumptions break, for example, types change after many iterations, it can always deoptimize (deopt) back to a lower tier. Examples of engine tiers include:

V8: Sparkplug, Maglev, and TurboFan [50].
SpiderMonkey: Baseline and WarpMonkey [39].
JavaScriptCore: Baseline, DFG, and FTL [20].

There are many possible engine differences, and they can be complex, but the key observation is that there are substantial architectural differences. These are the reasons why small code changes can matter a lot, and why we need benchmarked data for code patterns.

Category	Variant A	Variant B	Variant C
Execution	Interpreter	JIT	AOT
Parsing/compilation	Eagerly	Lazily	Both eagerly and lazily
Optimization approach	Minimal optimization	Profiled optimization	Aggressive speculative optimization
Code representation	AST execution	Bytecode	Native machine code
Type handling	Dynamic typing only	Type specialization	Speculative typing
De-optimization strategy	None	Partial deopt	Full deopt + recompile
Memory management	Reference counting	Mark-and-sweep GC	Generational GC

TABLE II: High-level architectural differences of JavaScript engines.

Table II lists some of the categories of JavaScript architecture and some of their variants. From this simplified list, it is evident that the same code can be executed in many different ways, while still accomplishing the same task. For example, something as simple as object properties (`{ a: "foo", b: "bar" }`) is complex, as described in a V8 blog post [3].

The flexibility in implementing the standard across engines leads to different trade-offs at each

decision that engine developers make, highlighting that there is likely a difference in energy use of smaller code patterns that reach the same result.

B. Sources of Efficiency Variability in JavaScript

As described above, even with identical code, efficiency variability can arise from the engine itself. For example, how garbage collection handles different allocation patterns, which heuristics it uses to optimize code, and how aggressively it does so.

Engines depend on observing runtime behavior, making speculative assumptions, proving invariants, etc. Functionally equivalent small code snippets that differ in structure or behavior can cause the engine to execute them along significantly different execution paths [17].

JavaScript is a dynamically typed language [33], meaning that a variable is not bound to a fixed type. It is therefore the engine's job to track and infer the type used during execution. When a variable maintains the same type over time, an engine can aggressively optimize it. On the other hand, introducing a mixture of types is likely to lead to deoptimization.

A common example of performance variability in coding is hoisting variables. For example, recomputing the same factor or reconstructing the same regex repeatedly within a loop is unnecessary, as it can be precomputed once before the loop. If an engine can correctly predict this, it can optimize it. On the other hand, if the developer attempts to optimize it themselves, they may hinder the engine's ability to make valid assumptions, resulting in negative returns.

Another factor is the predictability of control flows [36]. If the engine can accurately predict the program's branching behavior from runtime data, it can optimize substantially. For example, if the engine detects that a branch is always taken, it is beneficial to generate efficient machine code for that branch or to reorder the branching checks so that it checks for the expected branch before the unused branches. Two functionally equivalent implementations can yield different efficiency depending on how the engine predicts their paths.

JavaScript engines also include safety and security checks, which come at various costs [52, 44]. Safety checks are to prevent the program from doing something invalid or crashing. Security checks are intended to prevent malicious code from escaping restrictions or exploiting the engine in unintended ways.

JavaScript also provides multiple built-in methods to accomplish similar, if not identical, tasks. For example, searching in an array with `includes` compared to `indexOf`, transforming an

array using built-in methods such as `map` or `filter` compared to doing it manually using one of the many loop methods, or merging two objects using `Object.assign()` compared to the `...` spread operator.

These factors illustrate a subset of potential influences on energy use.

As a result, it is necessary to isolate small code snippets to study the impact on energy efficiency across JavaScript engines.

C. Energy Measurement Using RAPL

JavaScript is a language that predominantly runs on the CPU, except when executing GPU-accelerated workloads, such as rendering shaders or canvases. The benchmarks in my previous work [4] and this work will all run on the CPU and have no I/O. Like my previous work [4], measuring the energy consumption of such a program can be done using Running Average Power Limit (RAPL) [23] to record the CPU's energy consumption. There are other methods, such as devices connected directly to the power socket, but they do not offer anything to this project that will change the outcome, as shown by the work by Khan et al. They report a RAPL overhead of less than 1%, a 99% correlation, and a 1.7% mean absolute percentage error compared to a device directly connected to the power socket [28]. While it is true that RAPL reports consumption across all CPU cores, depending on domains used, the benchmarks will run on an isolated machine. If anything, measuring all CPU cores rather than just one is more accurate, since a garbage collector may be running on other cores.

The work of van Kempen et al. shows that memory contributes only a small fraction of a program's energy consumption, and that the majority of the cost it contributes is required to refresh memory cells for functional memory [27]. However, this work will still document the maximum Resident Set Size (RSS) for each execution as another parameter to help explain why one pattern may be more or less efficient, as it may involve a memory trade-off. Note that this is only the peak memory size and cannot reflect memory use over time during the program's execution.

The special-purpose CPU Model-Specific Register (MSR) reports the RAPL measurements. This thesis's benchmarks are run on Linux, where the special register can be accessed as a file at `"/sys/class/powercap/intel-rapl:0/energy_uj"` for the PKG domain. The domains are described below based on the Intel Manual [23], in the same way as my previous work [4]:

- **PKG**: The entire socket, i.e., all cores, integrated graphics, and the uncore components, which are the CPU parts outside of the cores, such as last-level cache, and ring interconnect.
- **PP0**: The CPU cores.
- **PP1**: Integrated graphics and uncore components, if applicable.
- **DRAM**: RAM attached to the CPU's memory controller - only supported in the server platform.

The measurements are reported in microjoules. The RSS is reported by the `/usr/bin/time -f "%M"` command and is reported in kilobytes.

D. Benchmarking and Measurement Challenges

For the reasons listed above, the benchmarks must use code that is not entirely optimized away by an optimistic engine, such as dead code elimination or loop collapsing.

Comparative analysis is likely unreliable when comparing a single-run measurement, given the many factors that can affect engine efficiency and the decisions it makes. Therefore, each benchmark is typically executed multiple times to account for variance [18].

The invocation of an engine on a benchmark file must be pinned to a single CPU core to minimize context switching and scheduling. Furthermore, the CPU frequency must be set to performance mode with turbo disabled to reduce noise. Performance mode locks the CPU to its highest frequency. Only a single benchmark is executed at a time, with no other programs running in the background besides the operating system and its required services, and a sleep period between benchmarks to reduce carryover effects from thermals, cache, etc.

Since we are dealing with JIT, warm-up behavior is expected; therefore, the methodology must include a mechanism to document the growth curve for each benchmark. It is not in this project's interest to eliminate warm-up, but rather to document its behavior and growth from end-to-end, as it is beneficial to know whether a given pattern starts out faster, then slows, and then accelerates again, etc. Furthermore, attempting to perform an unmeasured warmup across all engines is highly unreliable, as it would require communication between the JavaScript file and the benchmark runner performing the measurements, but not all engines support input or output. Furthermore, relying on a single-threaded JavaScript process to wait for communication from the benchmark runner script is unreliable due to synchronization issues. The alternative would be to perform the measurements within the JavaScript file, but not all engines support MSR access as

they do not support file reading. For these reasons, the warm-up behavior is documented rather than discarded.

Under identical hardware and engine versions, reproducibility is expected to be high, though variance should be kept in mind, as engines are not necessarily deterministic. The benchmark runner and the benchmarks are available on GitHub [5]. The benchmark runner differs from the one in my previous work [4] in several ways, including determining the iteration count for each engine-category combination and generating near-duplicate template files rather than wrapping them, as explained in Section 5-E.

E. Static Analysis and Linting

Static analysis is the analysis of source code without executing it. In JavaScript, ESLint [16] is a widely used tool for enforcing rules and providing customization, with over 100 million weekly downloads on NPM [15]. Therefore, this is a good tool for extending with rules to recommend code pattern replacements. Fundamentally, ESLint and many other linting tools work by converting the code into an AST, which is then evaluated for patterns. This thesis will explore whether ESLint rules can be created from energy-efficiency benchmark results. While many optimizations depend on runtime behavior and can vary with specific workloads and data types, this thesis will try to inform the developer when potential optimizations are detected, rather than strictly enforcing them.

Listing 1: ESLint rule example.

```
export default {
    meta: {
        type: "suggestion",
        docs: {
            description: "Prefer let over var",
        },
        fixable: "code",
    },
    create(context) {
        return {
            VariableDeclaration(node) {
                if (node.kind === "var") {
                    context.report({
                        node,
                        message: "Use let instead of var"
                    });
                }
            },
```

```
        };
    },
};
```

Listing 1 shows a rule written for ESLint that enforces `let` over `var`. It works by visiting every VariableDeclaration, and checking if the kind of node is `var`. Other node kinds are `let` and `const`.

A fixer can be provided to automatically replace `var` with `let` given that the user runs ESLint with the `--fix` flag, as demonstrated in Listing 2.

Listing 2: ESLint rule fixer example.

```
meta: {
    [...]
    fixable: "code",
},
context.report({
    [...]
    fix: fixer => fixer.replaceText(node, "let"),
});
```

F. JavaScript Performance Guidelines in Practice

Developers aim to improve their code's performance, and guidelines for doing so are widely available on forums, blogs, and in communities. However, the guidance is often derived from isolated experiments conducted in a single browser or runtime environment [26, 35, 46].

Therefore, the results on which these guidelines are based may be influenced by engine-specific behavior [26, 35, 46]. If someone tested it in Google Chrome (V8), it may behave differently in Safari (JavaScriptCore). This makes it hard for developers to generalize recommendations.

While existing guidelines based on a single engine can provide useful intuition, they just do not necessarily reflect how another engine will optimize that same code under otherwise identical conditions.

Online tools do exist that benchmark JavaScript code and provide insight into its performance [26, 35, 46]. However, they run in the user's device's browser environment, which cannot be generalized and may be affected by noise.

This further motivates the need for a controlled, cross-engine benchmarking of smaller code patterns. Isolating smaller code snippets and evaluating them under identical conditions across

engines enables assessment of whether recommended optimizations are beneficial for each engine.

5. Methodology

This chapter describes the methodology used to compare small code patterns with single sources of variation in a controlled environment across JavaScript engines.

A. JavaScript Engine Setup

The engines that this thesis will focus on are the same as in my previous work [4], listed below:

Name	Target Area	JIT	Version
V8 [51]	Google Chrome / Node.js	Yes	v15.1.6
SpiderMonkey (SM) [40]	Firefox	Yes	v152.0
JavaScriptCore (JSC) [20]	Safari	Yes	v314713
GraalJS [41]	Embedded in Java	Yes	v25.0.3
Hermes [9]	React Native	No (AOT)	v0.13.0
QuickJS (QJS) [49]	Lightweight/Embedded	No	v0.15.1
Moddable XS (XS) [38]	Microcontrollers	No	v8.2.0

TABLE III: Engines included in this study, downloaded via JSVU.

The reason for choosing the engines shown in Table III is that they are highly popular, powering the most widely used web browsers and backend environments, as well as popular specialized engines. There are over 100 JavaScript engines [29], and including more engines will yield diminishing returns globally, as they are less widely used. The JSVU tool by the Google Chrome Labs team is used to manage the engines [31]. The goal is not necessarily to compare engines to each other, but to find the best code pattern for each. By including multiple engines with different architectures, results might indicate some patterns that can be generalized.

B. Benchmark Selection

The benchmarks are selected from a variety of sources, including research papers, performance websites, forum posts, and browsing JavaScript documentation for similar functionality [1, 26, 35, 46, 53].

The benchmarks are assigned a category; e.g., ways to replace occurrences of a character in a string can be a category, with multiple benchmarks within it that all accomplish the same task.

Listing 3 and Listing 4 show 2 of the ways that string replacement can be accomplished. In this benchmark, it replaces all occurrences of an underscore (_) in a string with a dash (−).

Listing 3: String replacement method 1.

```
str.replace(/_/g, "-");
```

Listing 4: String replacement method 2.

```
str.split("_").join("-");
```

The full benchmark code for such a benchmark code would look as shown in Listing 5. The number of iterations is determined dynamically based on the engine, as described in Section 5-E.

Listing 5: String replacement method 3.

```
const str = "a_b_c_d_e_f_g_h_i_j_k_l_m_n_o_p_q_r_s_t_u_v_w_x_y_z";

let s = 0;

for (let i = 0; i < __ITERATIONS__; i++) {
    s += str.replace(/_/g, "-").length;
}

const out = s;
```

At first glance, the extra unused variables, arithmetic, and `.length` property access seems confusing. These are included to force the engines to do the work, as they otherwise tend to collapse the intended work into nothing, inspired by the *Blackhole* concept of Java Microbenchmark Harness [42]. Nonetheless, all benchmarks follow the same pattern, and the only source of difference is the method in the loop. While this may not reflect absolute costs, it will still show relative performance [18], i.e., which methods perform better or worse on each engine.

In total, 25 benchmark categories are included and defined based on the task that the pattern performs, as shown in Table IV, and all individual benchmarks within categories can be found on GitHub [5]. Every benchmark is manually tested to ensure consistent output, i.e., that multiple implementations produce the same result.

Category	Purpose	Benchmarks
access-char	Accessing a character in a string from given index	2
array-max	Finding the maximum value in an array	3
even-odd	Determining if a number is even or odd	2
filter	Filtering values from an array	2
flatten-array	Flattening an array	3
function	Defining a function	3
json	Hoisting the parse call of a JSON string	2
lookup	Looking up in an object vs in a map	2
loop-arr-length	Hoisting the length of an array in a loop	3
map-reduce	Apply a function to all elements and accumulate into a single value	2
math-floor	Round a number down to the nearest integer	4
nested-lookup	Lookup in 2 arrays with a nested loop vs a map	2
number-to-string	Converting a number to a string	4
object-iteration	Looping over an object	2
object-merging	Merging objects into a single object	2
object-shape	Defining object vs empty object	2
polymorphic	Array with 2 types versus 2 arrays of individual types	2
prototype	Defining a function vs defining a prototype function	2
regex	Hoisting regex	3
replace-loop	Replace occurrences of a character in a string many times.	5
replace-size	Replace occurrences of a character in a longer string	5
search-string	Searching for a string in a string	2
string-from-array	Transforming an array of characters into a string	4
substring	Searching for a substring of length 2 in a string	3
sum-array	Sum the values of an array	2
Total		68

TABLE IV: Benchmark categories and their intended evaluation focus.

C. Hardware and System Configuration

All benchmarks are run on the same dedicated machine, one at a time. The machine has the specifications shown in Table V, and is the same machine used in my previous work [4].

Component	Specification
CPU	Intel Core i7-8700 @ 3200 MHz (800–4600 MHz)
GPU	NVIDIA GeForce GTX 1070
Memory	16GiB DDR4 (2x8GiB) @ 2666 MHz
Motherboard	ASUS TUF B360M-PLUS GAMING
Operating System	Ubuntu 24.04.3 LTS

TABLE V: Hardware specifications of the dedicated desktop machine.

D. Metrics

The energy consumption, runtime, and memory of each sample are measured and saved individually.

The energy is measured in microjoules using RAPL, the runtime is measured in seconds using `date`, and the memory is measured using `usr/bin/time -f "%M"`, as described in Section 4-C.

- *Energy (μJ)*: Microjoules delta from the PKG domain.
- *Time (s)*: Seconds of real wall-clock time.
- *Memory (kb)*: Kilobytes of RSS.

E. Execution Procedure

At this level of code granularity, i.e., small patterns with a single source of variation and not algorithms, and given the vast differences in behavior across engines, using the same iteration counts for all engines proved futile since some engines finish execution almost instantly, and others attempt execution until they run out of memory.

Therefore, the benchmark runner is a bash script that calibrates the number of iterations per engine per category by sweeping for N_{max}. In this context, sweeping means repeatedly running the benchmark while varying a single parameter. The N_{max} is the maximum number of iterations for an engine and category combination. It accomplishes this by starting at N=1024 (number of iterations) and doubling until the time to execute the baseline benchmark exceeds 10 seconds, at which point it uses the previous value. Therefore, the used N_{max} value is $N \in 1024 * 2^k$. The other benchmarks in the same category use the same N value as the baseline benchmark. For each benchmark in a category, it executes the benchmark with values of N doubling from 1024 up to the calibrated N_{max} value, as shown in Listing 6:

Listing 6: Loop from N to N_{max}.

```
for N = 1024; N <= N_max; N *= 2:
    run benchmark
```

Each benchmark is sampled 10 times per value of N, i.e., 10 samples before doubling N.

Some engines do not accept input; instead, the bash script uses a template file with a variable to be replaced: `__ITERATIONS__`. This variable in the template file is replaced and used to generate many identical files, except for the value of the variable that replaces `__ITERATIONS__`. This eliminates the need to pass variables to every engine, as they instead run a file that hardcodes the required value.

Each benchmark is executed individually on a single engine, with no other benchmarks running concurrently. The procedure uses a similar system configuration as in my previous project [4]:

1) Added RAPL overflow detection and correction by checking if the delta between before and after is negative, and then adding `MAX_ENERGY` from `sys/class/powercap/intel-rapl:0/max_energy_range_uj`.
   This detects only one overflow, but the range is so large that it would take multiple hours of a single benchmark run to overflow more than once, which is not observed in any benchmark.
2) Added one second of sleep in between each benchmark to reduce noise and throttling.
3) Pinned the invocation of `./engine` to a single CPU core using taskset to avoid scheduling and context switching. Uses core 1, as core 0 may be used internally by the operating system.
4) Set the CPU governor to performance mode using
   `sudo cpupower frequency-set -g performance`
   to ensure stable CPU frequency.
5) Disabled turbo using
   `echo 1 | sudo /sys/devices/system/cpu/intel_pstate/no_turbo` so that the CPU runs on a more predictable fixed frequency.

Warm-up behavior is not eliminated for reasons mentioned in Section 4-D, but instead measured.

6. RESULTS

This chapter outlines what the data shows, while the next chapter will discuss what the results mean. The full dataset and steps to reproduce are available on GitHub [5].

A. Dataset Overview

The raw data is collected in a `.csv` file and is not processed by the bash script. All processing is performed in Python with scripts available on GitHub. Since each sample (engine x benchmark x N) is repeated 10 times, the results are summarized using the median to most accurately reflect a normal run. The median is chosen over the average because it is more robust to outliers, which are expected in this environment. Variability will be reported using Interquartile Range (IQR).

Category	**Test**	**Engine**	**N**	**Sample**	**Energy (μJ)**	**Time (s)**	**Mem (KB)**
access-char	baseline	v8	1024	1	388488	0.0204838	28800
access-char	baseline	v8	1024	2	405456	0.0202289	28544
access-char	baseline	v8	1024	3	386290	0.0206003	28544
access-char	baseline	v8	1024	4	396667	0.0200686	28416
access-char	baseline	v8	1024	5	393004	0.0202918	28544
access-char	baseline	v8	1024	6	395018	0.0203822	28672
access-char	baseline	v8	1024	7	415038	0.0203126	28672
access-char	baseline	v8	1024	8	395690	0.0199227	28544
access-char	baseline	v8	1024	9	394896	0.0199567	28928
access-char	baseline	v8	1024	10	390136	0.0202877	28160
access-char	baseline	v8	2048	1	412535	0.0208113	30848
			... 65,788 entries ...				
sum-array	reduce	xs	65536	10	140290351	7.51319	53120

TABLE VI: Sample of benchmark results across engines, tests, and iterations (N).

Table VI shows the `.csv` data format and what an entry includes. The data contains 65,800 entries, obtained by running 10 samples of 68 benchmarks across 25 categories on 7 engines. Each engine and category has a maximum N input size range that starts at 1024 and doubles, as described in Section 5-E.

B. Overall Engine Differences

Figure 1 shows a heatmap of each engine and each benchmark category, where each row is a benchmark category, and each column is a JavaScript engine. The cells show the median

relative energy consumption for that benchmark category and engine combination, compared to the median of the most efficient test in that category, for that engine. This means that engines are not compared to other engines in this map. These values are based on the efficiency at max_N, which could be considered the steady state. This is an attempt to most accurately reflect the real-world characteristics of long-term optimization by reducing the fixed overhead and letting the engines optimize. The colors range from 1x (purple) to 10x (yellow). Anything above 10x is clipped to the same color, so the least efficient results do not skew all the colors, specifically JavaScriptCore's 2051x energy consumption for the *replace-loop* category.

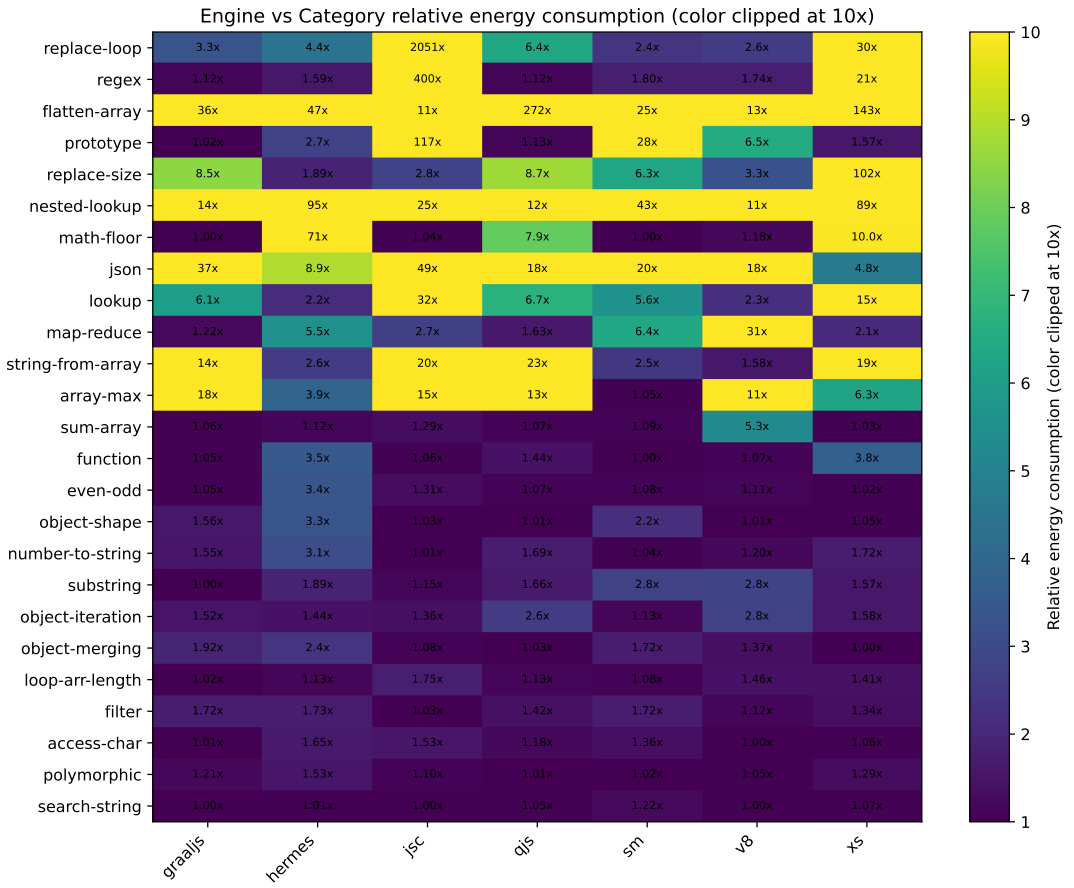


Fig. 1: Median relative energy consumptions of each engine for each benchmark category compared to the most efficient benchmark for that category.

For example, the V8 engine's efficiency at finding the maximum value in an array can be up to 11 times less efficient, depending on the implementation. There are 3 included implementations of maximum value:

Listing 7: Baseline (Built-in max() method).

```
max = Math.max(...arr);
```

Listing 8: For loop.

```
let max = -Infinity;

for (let j = 0; j < arr.length; j++) {
    if (arr[j] > max) {
        max = arr[j];
    }
}
```

Listing 9: Same as baseline but with prototype.apply.

```
max = Math.max.apply(null, arr);
```

For the V8 engine, the `for` loop is the most efficient, the `Math.max.apply()` is a close second at 1.18 times less efficient than the for loop, while the `Math.max()` method is 11.04 times less efficient than the for loop.

Some rows, such as the *search-string* category, perform similarly across all implementations across all engines. A category like this is unlikely to be a good target for optimization. However, there are also rows, such as the *flatten-array* category, that have another implementation that performs much worse across all engines, with efficiency ranging from 11 to 272 times worse. This makes sense when considering which implementations are compared:

Listing 10: Baseline (reduce).

```
arr.reduce((acc, x) => acc.concat(x), []);
```

Listing 11: Built-in flat() method.

```
arr.flat();
```

Listing 12: Built-in concat() method on a spread array.

```
[].concat(...arr);
```

In all cases, the `reduce()` method is the least efficient. This makes sense because `reduce()` is more of a general multi-purpose method that accepts a function, while `flat()` and `concat()` are a bit more specialized. Implementations like this are still included to showcase engine optimization or a lack of optimization. However, depending on the engine, `flat()` can be both more and less efficient. For example, on Hermes and V8, `flat()` is 2.00 and 1.46 times less efficient than `concat()`. Conversely, on JavaScriptCore and GraalJS, `concat()` is 2.44 and 1.54 times less efficient. This goes to show that while some implementations are always worse for all engines, the most efficient one can depend on the engine.

When counting yellow squares, Hermes has the fewest, with 3, and JavaScriptCore has the most, with 9. This could be interpreted in multiple ways. One interpretation is that Hermes is less volatile or better at optimizing for other implementations that accomplish the same task, and that the choice of implementation matters more in JavaScriptCore. Another interpretation is that JavaScriptCore is extremely good at optimizing certain things, which makes the other implementations seem terrible by comparison. It is unfair to compare engines in this way, since their relative energy consumption is assessed at the intra-engine level. Furthermore, the goal is not to find the most efficient engines; therefore, this line of thinking will not be explored further in the discussion chapter in Section 7.

C. Distribution of Relative Energy Consumption

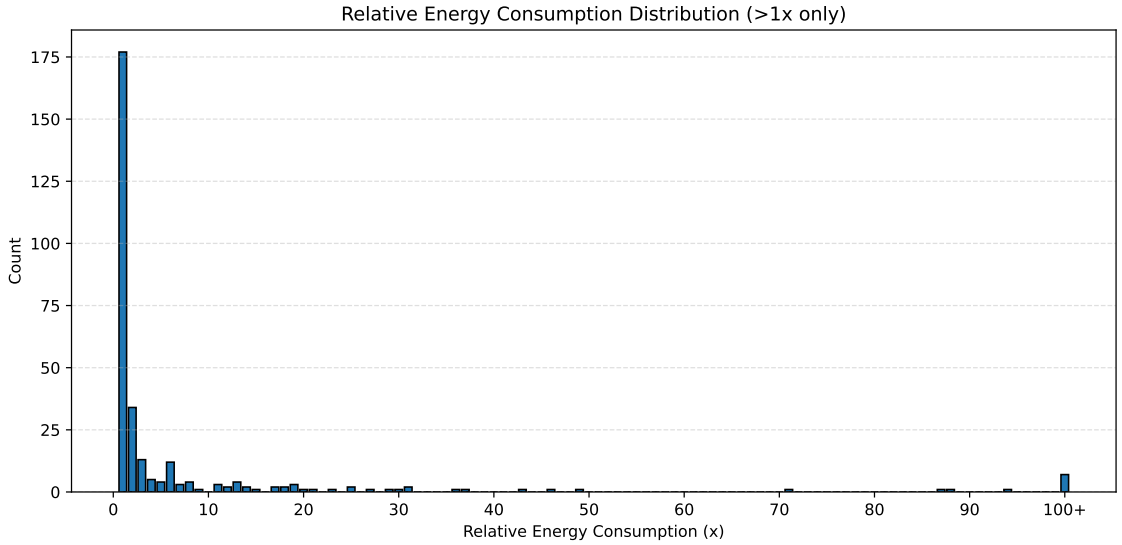


Fig. 2: Counts of relative energy consumption where each bar represents a range of 1.

Figure 2 shows the distribution of the relative energy consumption. One bar represents a range of 1, i.e., the first bar is the range of 1 to 2 times relative efficiency. The value of 1 is excluded, meaning that the count of entries where it is the best implementation is not counted. There is a small spread of other bars, but they individually pale in comparison to the first bar. When considering all bars above 2x as a single unit, it is clear that there are many instances where optimization is possible. The last bar includes all values above 100x. This first bar dominates the graph with just over 175 entries, and therefore Figure 3 zooms in on this first bar.

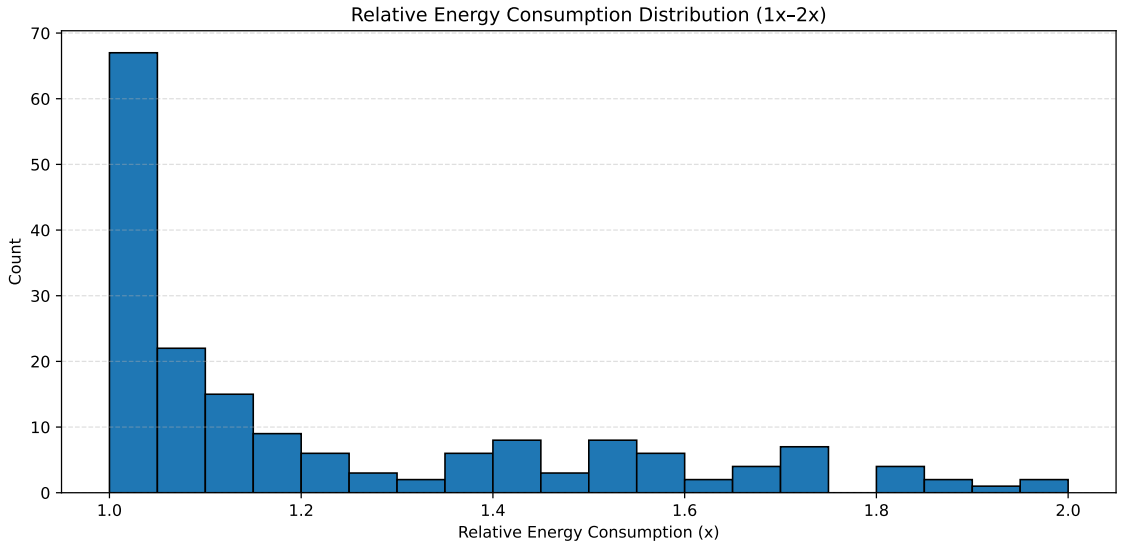


Fig. 3: Counts of relative energy consumption between 1 and 2x, where each bar represents a range of 0.05.

Zooming in on the first bar shows a good spread of entries consuming between 1 and 2 times as much energy. This shows that the first bar is not dominated by values close to 1. In other words, there are many data entries where optimization can be up to 2 times as efficient.

D. Medians and Variability

The following 2 figures display medians through blue bars, and variability through black lines. The black lines are IQR, with the bottom whisker being 25%, and the top whisker being 75%.

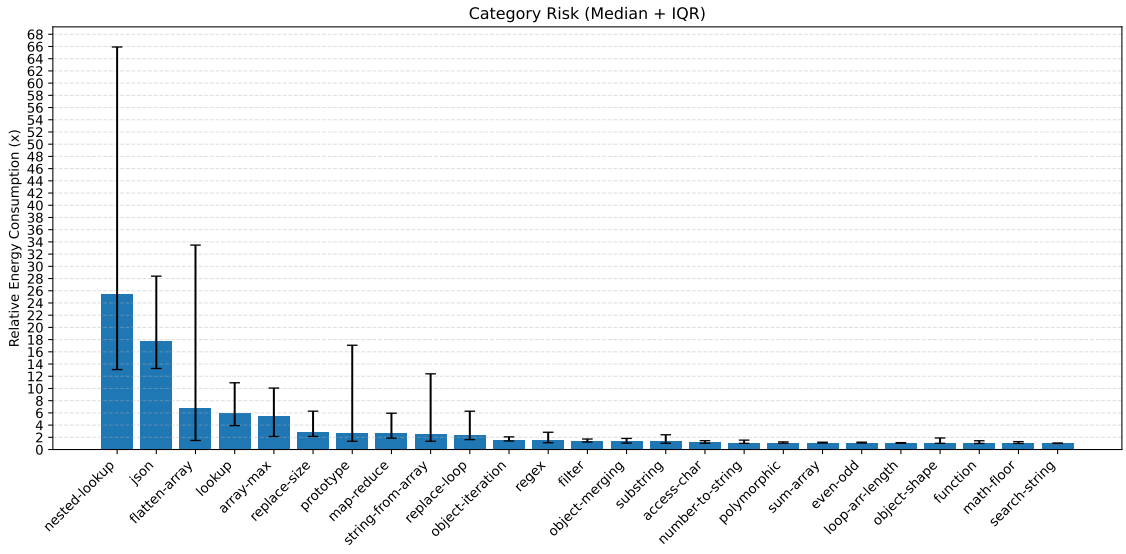


Fig. 4: Medians and variability for all benchmarked categories.

Figure 4 shows the median and variability of every benchmarked category. The size of the blue bars indicates that optimizing categories such as *nested-lookup* and *json* has much greater potential for energy-efficiency impact than categories such as *math-floor* and *search-string*. The black lines indicate significant variability, but this is mostly because some optimizations perform better on one engine and worse on another. Despite this, some categories are more stable than others, such as *lookup* and *replace-size* having relatively small variability compared to *nested-lookup*, *flatten-array*, and prototype.

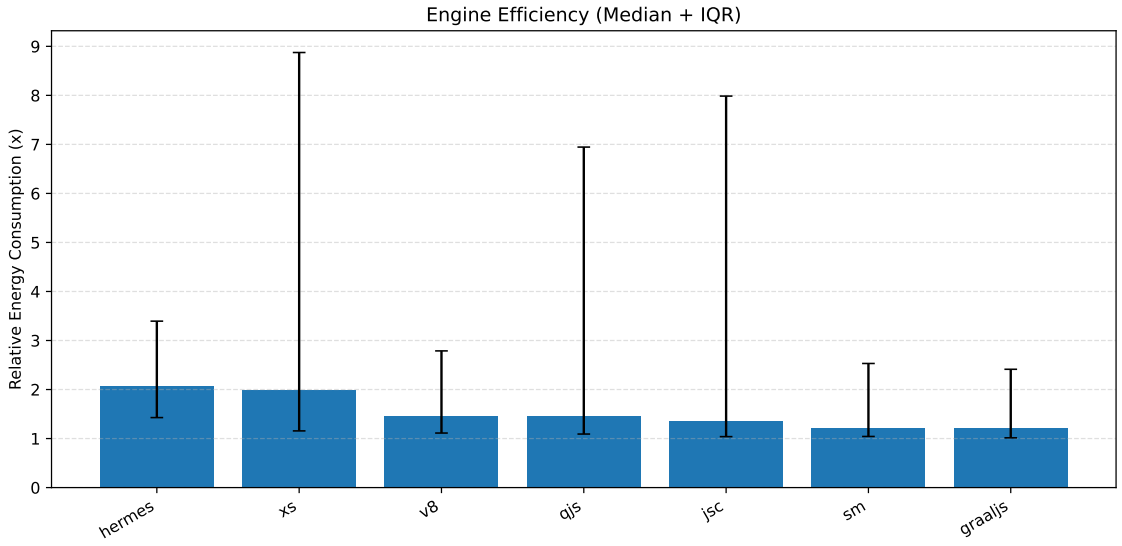


Fig. 5: Medians and variability for all included JavaScript engines.

Figure 5 shows the median and variability of the JavaScript engines. The median relative energy consumption across all engines is at or below 2x, with SpiderMonkey and GraalJS hovering near 1x. XS, JavaScriptCore, and QuickJS, in that order, have the most variability.

All of this indicates that statistical analysis of smaller code patterns can be beneficial for optimizing energy efficiency.

E. In-depth Look at V8

I select V8 as the basis for an in-depth look, given its popularity and high median, which indicate it has the greatest potential. This means that V8 is the focus for ESLint rules in Section 8. While this section focuses on V8, every figure for every engine and every benchmark is pre-computed and available on GitHub [5], as there is too much to reasonably include in the appendix. However, Appendix A includes 1 figure for every engine that shows the engine's worst-case relative energy consumption for every category.

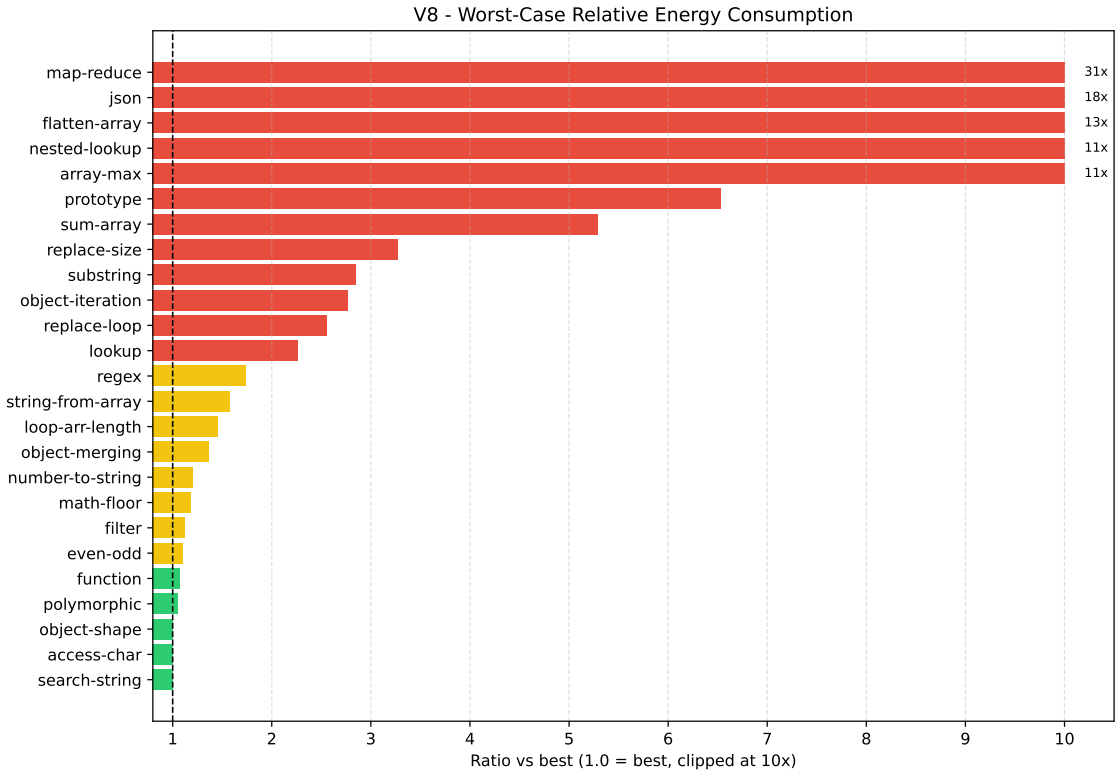


Fig. 6: Worst-case relative energy consumption for every category for the V8 engine.

Figure 6 shows the worst-case relative energy consumption for every benchmark category. The green colors indicate values below 1.1x, yellow indicates values below 2x, and red indicates values above 2x, with clipping at 10x to keep the figure readable. 5 categories are green, 8 categories are yellow, and 12 categories are red. This is roughly the same pattern across all engines: mostly red, with not much green.

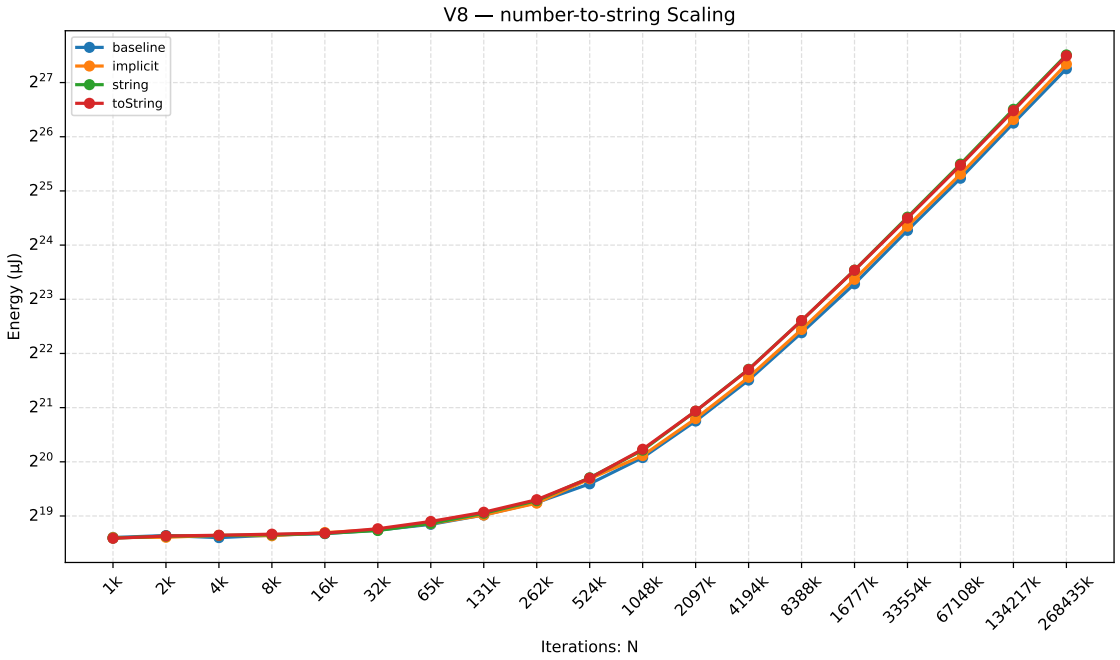


Fig. 7: Median of all samples for V8's *number-to-string* category.

Figure 7 shows the growth of a category, *number-to-string*, where all 4 implementations are near each other, follow the same shape, and roughly remain in the same relative positions throughout all values of N, i.e., no drastic changes. This might be what one expects of a modern engine for a task as simple as converting a number to a string. Furthermore, after about 1048k, when N doubles, the energy also roughly doubles. The following 4 ways are included:

Listing 13: The 4 included ways of converting a number to a string.

```
baseline: `${i}`;
implicit: "" + i;
string:   String(i);
toString: i.toString();
```

While some results are as expected, others are not. Figure 8 shows a comparison for hoisting the length of an array when looping over an array using a traditional `for` loop. In this example, hoisting the length variable increases the energy cost: `const len = arr.length;`. However, inlining the variable consumes as much energy as writing the loop the traditional way.

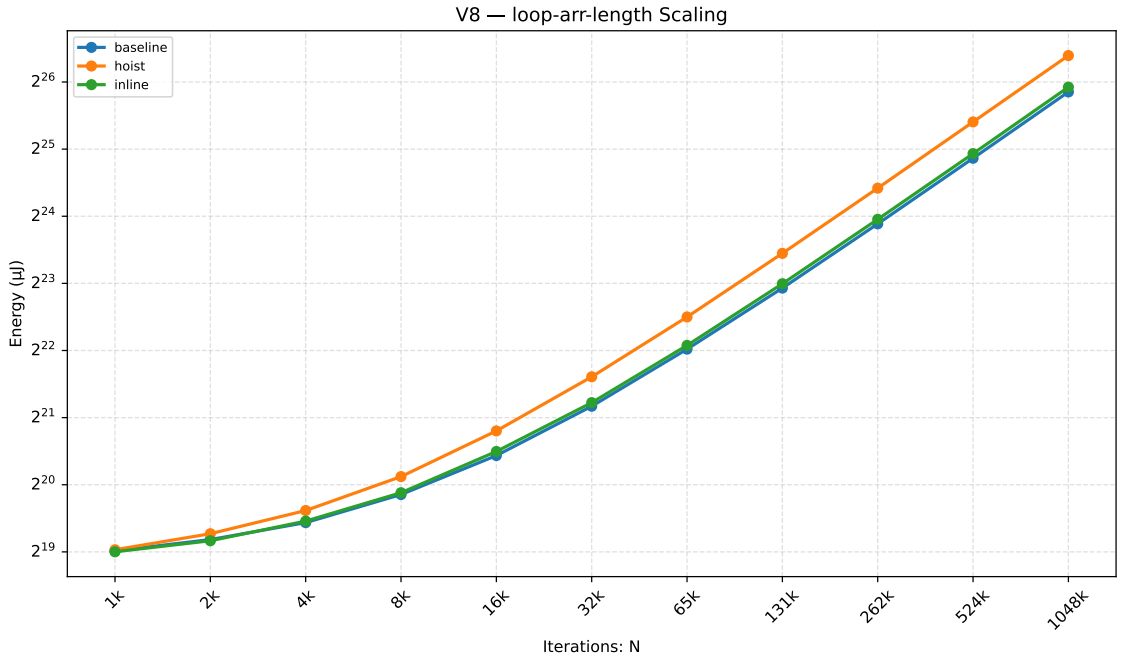


Fig. 8: Median of all samples for V8's *loop-arr-length* category.

Listing 14: Baseline (traditional way).

```
for (let j = 0; j < arr.length; j++) { ... }
```

Listing 15: Hoisting a length variable.

```
const len = arr.length;
for (let j = 0; j < len; j++) { ... }
```

Listing 16: Inlining a length variable.

```
for (let j = 0, len = arr.length; j < len; j++) { ... }
```

This pattern does not generalize. For example, on the XS engine, hoisting and inlining both perform better than the baseline. In fact, this general pattern of hoisting does not even generalize to the V8 engine. At its worst, hoisting consumes 1.46 times as much energy, and inlining consumes 1.05 times as much.

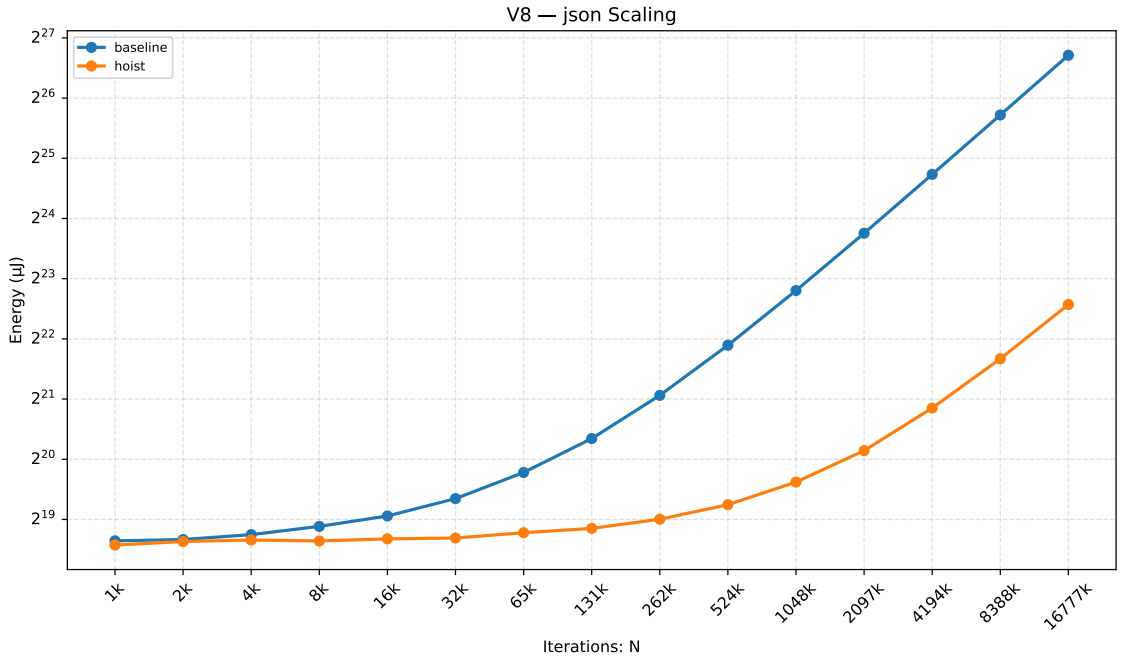


Fig. 9: Median of all samples for V8's *json* category.

Figure 9 shows that hoisting drastically decreases the energy consumption. In this case, the variable being hoisted is a call to the `JSON.parse()` method. At its worst, not hoisting consumes 17.63 times as much energy.

Listing 17: Hoisting the call to `JSON.parse()`.

```
const config = JSON.parse(jsonString);
for (let i = 0; i < __ITERATIONS__; i++) {
    ...
}
```

Listing 18: Not hoisting the call to `JSON.parse()` (baseline).

```
for (let i = 0; i < __ITERATIONS__; i++) {
    const config = JSON.parse(jsonString);
    ...
}
```

This indicates that generalized rules cannot be made based on a pattern such as hoisting; they also depend on context. In other words, rules must be very specific.

7. Discussion

This chapter discusses research questions 1-3, as research question 4 is based on ESLint rules introduced in the next chapter. The results from Section 6 showed that there are significantly measurable differences between small code patterns across JavaScript engines.

A. RQ1: Choice of Engine

RQ1: ”How does the choice of JavaScript engine affect the energy consumption of small JavaScript code snippets?”
The data from Section 6 shows that while the choice of JavaScript engine does have an impact, it is not the only factor. Some of the categories, such as *flatten-array* and *nested-lookup*, have other implementations with significantly different energy consumption. In contrast, other categories used roughly the same amount of energy regardless of implementation across all engines. This is the case for *search-string*, *polymorphic*, and *access-char*.

When the choice of JavaScript engine does have an impact, it can be huge. JavaScriptCore has relative energy consumption values of up to 400 and 2051 times for the categories *regex* and *replace-loop*, respectively, whereas the same categories have relative energy consumption values of up to 1.8 and 2.4 times, respectively, for the SpiderMonkey engine. This does not necessarily mean that SpiderMonkey is more efficient, as these numbers are relative to the engine’s own highest efficiency. In fact, JavaScriptCore’s lowest consumption was 1.92 J at N=16777216, while SpiderMonkey’s lowest was 152.73 J at the same N. These most efficient performances, however, are for different implementations.

When looking at the median and variability from Section 6-D, Hermes and XS have a much higher median than the rest at around 2x, while the others are below 1.5x. XS, QuickJS, and JavaScriptCore show much greater variability than the rest, with ranges around 7-8x, while the others are around 1-1.5x. Therefore, the choice of JavaScript engine can have an unpredictable, but measurable, impact depending on the code executed.

B. RQ2: Engine-Dependent Optimizations

RQ2: ”Can a different optimization pattern that reduces energy consumption in one JavaScript engine have no effect or an adverse effect on another engine?”
As outlined in Section 6-B, a pattern that is more efficient on one engine acts the opposite way

on another. This is also the conclusion of my previous work, but at the scale of small code patterns rather than algorithms [4].

It is not feasible to pinpoint the exact reason for such behavior, as engines are complex. However, as described in Section 4, engines are inherently structured differently; therefore, it is not surprising that patterns vary by engine. The background chapter is included to give intuition for why there are such varied discrepancies between engines that implement the same standard, and is not meant to be used as a fact sheet for explaining specific engine behavior. While this result might seem obvious, it is still important to confirm, as it proves that any static analysis rules must be on a per-engine basis. The results confirm that the patterns are not consistent across engines and can have opposite effects depending on the engine.

Studies that evaluate energy efficiency based on a single engine should be careful about generalizing their findings.

Therefore, I propose 3 different categories of rules:

1) Universal rules
2) Engine-specific rules
3) Context-dependent rules

Universal rules are more efficient across all engines, such as the *object-shape* category, where defining the shape of an object consistently consumed the least amount of energy on all engines. One could argue that this intuitively makes sense, but based on the other results, such as hoisting, it is not always trivial.

Listing 19: No object shape.

```
1 const obj = {};
2 obj.a = 1;
```

Listing 20: Defined object shape.

```
1 const obj = { a: 1 };
```

Engine-specific rules apply to one or more engines, while other engines have different rules for the same category. This could be ways to flatten an array, as outlined in Section 6-B. While some benchmarks do imply that one pattern is more efficient than another, they sometimes fall

within margins of error, for example, V8 `indexOf()` vs `includes()` at 1.001x efficiency difference.

Context-dependent rules require deeper static analysis or more benchmarking, such as the hoisting outlined in Section 6-E.

C. RQ3: Static Analysis Rules

RQ3: ”Can identified optimization patterns be expressed as static analysis rules for automated detection?”
In order to implement such rules for cases where it is evident that a pattern is more efficient, universally, or for a specific engine, it must be feasible to detect such patterns. Detecting certain patterns can be achieved as described in Section 4-E by evaluating the source code without executing it. Many of the included categories are syntactically simple, such as using different built-in methods. Others are more involved, such as performing a nested lookup with maps rather than arrays. Optimizations based on JIT compilation or hidden runtime effects cannot be statically analyzed as they are inherently dynamic and require execution. Therefore, at least a subset of the observed optimization patterns can be reliably expressed as static analysis rules. Consequently, static analysis cannot replace runtime benchmarking; rather, it serves as a complementary mechanism.

Section 8 outlines how rules are implemented.

D. Limitations

This thesis is based on a single version for each engine, despite prior work showing that versions can have a significant impact. I have chosen this limitation because it is in the interest of developers and consumers to update their engines anyway, such as for safety and security reasons. Because of this, once the ESLint rules are written, all one needs to do is re-benchmark the new versions and change which rules apply, if any.

There are an unknown number of possible categories and implementations out there; this thesis’s 25-category set is but a small subset of possibilities and does not necessarily reflect real-world relative energy consumption.

Benchmarks are intentionally designed to isolate specific, small differences in source code while including as little work as possible, so that the engines do not optimize the work away. It is

possible that some engines have optimized away code for certain implementations in certain categories, while others have not.

8. ESLint Rules

Rules are created for the V8 engine based on its benchmarks. They are defined using ESLint to identify patterns and suggest replacements. Not every included category is implemented as a rule, as it must be identifiable without executing the source code and have consistent impacts on energy use. All the implemented ESLint Rules for the V8 engine can be found in Appendix B.

A. *Defining a Rule*

The *flatten-array* category is used as an example of how to define an ESLint rule. Based on my findings, flattening an array using `reduce(concat())` is the least efficient, `arr.flat()` the second least efficient, and `[].concat(...arr)` the most efficient. Therefore, a rule must be able to detect the 2 less efficient patterns.

Listing 21: ESLint rule for flattening an array in V8.

```
CallExpression(node) {
    // arr.flat()
    if (node.callee.type === "MemberExpression" && node.callee.property.name ===
        ↪ "flat") {
        context.report({
            node,
            message: "Use [].concat(...arr) instead of flat() for up to 1.46x better
                ↪ performance",
            fix(fixer) {
                const sourceCode = context.sourceCode;
                const obj = sourceCode.getText(node.callee.object);

                return fixer.replaceText(node, `[].concat(...${obj})`);
            },
        });
    }

    // arr.reduce(concat)
    if (node.callee.type === "MemberExpression" && node.callee.property.name ===
        ↪ "reduce") {
        const [callback] = node.arguments;

        if (callback && callback.type === "ArrowFunctionExpression") {
            const body = callback.body;

            if (
```

```
                body.type === "CallExpression" &&
                body.callee.type === "MemberExpression" &&
                body.callee.property.name === "concat"
            ) {
                context.report({
                    node,
                    message: "Use [].concat(...arr) instead of reduce+concat for up
                        ↪ to 13x better performance",
                    fix(fixer) {
                        const sourceCode = context.sourceCode;
                        const obj = sourceCode.getText(node.callee.object);

                        return fixer.replaceText(node, `[].concat(...${obj})`);
                    },
                });
            }
        }
    }
},
```

Listing 21 shows the ESLint rule that can detect the 2 less efficient patterns of flattening an array in V8, and includes a fixer function that replaces them with the more optimized pattern. Note that this is very general, as the fixer just replaces the node with a predefined pattern. If the pattern to replace is more complex, these fixers should not be applied automatically. Fortunately, ESLint is built for this. ESLint, when executed without any parameters, warns instead of automatically applying the fixer. When executed with the `--fix` flag, it will attempt to use the fixer functions. The warning messages include the potential to reduce energy consumption, as indicated by the benchmarks, to inform the developer.

Line 1, CallExpression, is a function that is being called. This means that this ESLint rule will be checked every time a function call is seen in the AST. Immediately after, in both cases on lines 3 and 17, there is a check for MemberExpression, which is property access such as `object.property`. This is because the rule is looking for `arr.flat()` and `arr.reduce()`. The second case requires more code, as the pattern is more complex. Lines 18-20 retrieve the callback body of the `reduce()` call, and check on lines 23-27 if it includes a `concat()` call. In both cases, it includes a `fix(fixer)` call that can replace the node with the more efficient pattern.

A rule like this is not perfect. There are multiple AST shapes for equivalent logic, for example `arr.reduce((a, b) => { return a.concat(b); }, []);` is not detected as this

uses a block body instead of a concise body for the `reduce` method. Of course, a rule can be expanded to include checks for that as well.

Listing 22: Code before applying ESLint rules.

```
1  arr.flat();
2
3  arr.reduce((a, b) => a.concat(b), []);
4
5  [].concat(...arr);
```

Listing 23 shows the code before ESLint rules are applied, as well as how the code will remain if the `--fix` flag is not provided. In that case, the developer would have to change it themselves based on the rule's informative message. Listing 23 shows the code after the fixes have been applied.

Listing 23: Code after applying ESLint rules.

```
1  [].concat(...arr);
2
3  [].concat(...arr);
4
5  [].concat(...arr);
```

The warnings can be received by running ESLint on a file, with output as seen in Figure 10.

```
>> npx eslint test.js

C:\Users\ander\Downloads\AAU\AAU VPN P10\ESLint\test.js
   9:1  warning  Use [].concat(...arr) instead of flat() for up to 1.46x better performance       energy-rules/flatten-array
  11:1  warning  Use [].concat(...arr) instead of reduce+concat for up to 13x better performance  energy-rules/flatten-array
  15:1  warning  Avoid Math.max(...arr), use loop-based max                                        energy-rules/array-max
  17:1  warning  Avoid Math.max.apply(null, arr), use loop-based max                               energy-rules/array-max
  19:1  warning  Avoid JSON.parse() inside loops                                                   energy-rules/json-parse-loop
  24:1  warning  Avoid JSON.parse() inside loops                                                   energy-rules/json-parse-loop
  29:1  warning  Avoid JSON.parse() inside loops                                                   energy-rules/json-parse-loop
  33:1  warning  Avoid JSON.parse() inside loops                                                   energy-rules/json-parse-loop
  37:1  warning  Avoid num | 0. Use Math.floor(num)                                                energy-rules/math-floor
  38:1  warning  Avoid ~~num. Use Math.floor(num)                                                  energy-rules/math-floor
  39:1  warning  Avoid parseInt(num). Use Math.floor(num)                                          energy-rules/math-floor

✖ 11 problems (0 errors, 11 warnings)
  0 errors and 2 warnings potentially fixable with the `--fix` option.
```

Fig. 10: ESLint warnings shown from a command terminal where ESLint is executed.

When used in combination with other extensions, such as VSCode's Error Lens, the warnings can be shown to the developer while they are coding, as shown in Figure 11.

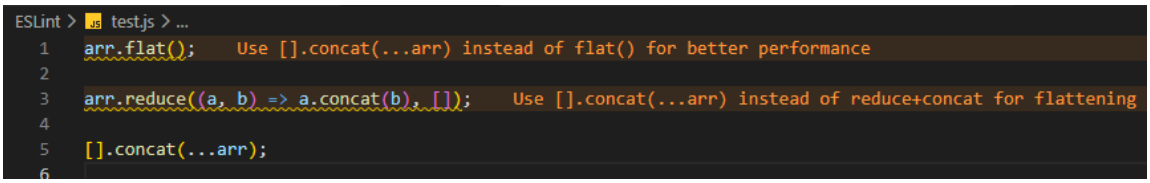


Fig. 11: ESLint warnings shown from VSCode's Error Lens extension.

9. Conclusion

This work compares the energy efficiency of 7 JavaScript engines across 25 benchmark categories, totaling 68 benchmarks, using 65,800 data entries. Previous work has focused on algorithms, whereas this work focuses on smaller code patterns with limited sources of variation. The purpose of this is to derive static analysis rules that guide developers in making their code more energy-efficient. All 7 engines were benchmarked under the same controlled conditions, but are not meant to be compared to each other, as it has already been shown that engine efficiency varies significantly. For this reason, varying numbers of iterations were determined and used for each engine-benchmark combination to push each engine to steady state. Because of this, the data allow for the deduction of per-engine rules to optimize energy efficiency.

For research question 1, these findings confirm that engine choice matters. The impact of the engine is not easily predictable, but it is measurable. Therefore, it is important to benchmark rather than assume when writing rules for static analysis. Some engines show little difference in certain categories while showing dramatic differences in others. It is not exactly determined why the engines behave as they do, since the goal and important thing is the outcome. The engines do vary in median and variability, meaning that if these findings generalize, some engines can be expected to have less drastic relative energy consumption than others.

For research question 2, these findings confirm that while some categories have a single implementation that dominates across all engines, the most efficient implementation for a category depends on the engine on which it is executed. This means that other studies and tools that base their suggestions on a single engine must use the same engine that the developer uses to execute the code. Otherwise, they may receive incorrect suggestions, which could make the code less efficient.

For research question 3, it is shown that patterns can be detected statically, but it is not always trivial to automatically replace the code with a more efficient pattern. Furthermore, there are

multiple ways to construct the code for the same pattern, which requires the rule to check for equivalent logic, for example, when using concise bodies versus block bodies in an arrow function. ESLint has been shown to be a feasible way to implement and use such rules, both for warning messages and auto-fixing when applicable. Therefore, I must conclude that it is possible to build practical tooling based on empirical benchmark data.

The findings show that energy-efficient JavaScript development can be partially guided by static analysis, provided the rules are based on per-engine analysis. From this work, I view static analysis as a supporting tool for decision-making, not as a perfect replacement. These findings, combined with previous work, show that energy efficiency depends heavily on the code's context and execution environment, so each developer must validate the measurements.

A. Future Work

This work has continued to define the state of the art in benchmarking JavaScript engines, but the process is far from over.

TypeScript - It is interesting to research the impact of transpiling TypeScript to JavaScript, i.e., if the patterns remain the same, or the TypeScript type checking makes assumptions for some patterns and not for others.

Benchmarking applied rules - Future research could use the most efficient patterns and apply them to existing algorithms or real-world projects, and compare the energy efficiency of the before with the after. This can help determine and validate the impact of these small patterns in the real world.

More of everything - It is, of course, always possible to include more engines, include versions, include more benchmarks, more implementations of categories, etc.

Generate rules automatically - I have not researched the feasibility of generating rules automatically, but it should, at the very least, be possible to assign pre-written rules to engines, based on continued benchmarking when new versions are released, so that developers can be informed with up-to-date intel.

Validating the work in the benchmarks - As mentioned in Section 7-D, it is possible that some engines have not fully done the work as intended, despite the numbers and graphs showing growth. Therefore, it might be important to include validation of the work in the benchmark itself, although this adds more overhead. Furthermore, it is debatable whether this is important,

since the engine's ability to optimize dead work relative to another engine could be seen as a real-world trait that should be embraced. However, under this assumption, rules must be very specific and not generalized. This is especially interesting because the work of Couto et al. finds that while individual refactors are consistently more beneficial than harmful, combining multiple refactors makes them harmful [6].

GLOSSARY

AOT
: Ahead-of-Time. 11, 12

AST
: Abstract Syntax Tree. 11, 16, 39

CLBG
: Computer Language Benchmark Game. 9

ICT
: Information and Communication Technology. 5

IQR
: Interquartile Range. 23, 29

JIT
: Just-in-Time. 8, 12, 15

MSR
: Model-Specific Register. 14, 15

RAPL
: Running Average Power Limit. 14, 21, 22

RSS
: Resident Set Size. 14, 15, 21

SCI
: Software Carbon Intensity. 5

REFERENCES

[1] Various authors. *JavaScript String replace vs replaceAll*. Accessed: 19-05-2026. URL: https://stackoverflow.com/questions/67296652/javascript-string-replace-vs-replaceall.

[2] Fabrice Bellard and Charlie Gordon. *QuickJS Javascript Engine*. Accessed: 01-06-2026. URL: https://bellard.org/quickjs/quickjs.html.

[3] Camillo Bruni. *Fast properties in V8*. Accessed: 19-05-2026. URL: https://v8.dev/blog/fast-properties.

[4] Anders Nyby Christensen. "Energy Efficiency of JavaScript Engines: A Multi-Version, Multi-Benchmark Study". In: *Aalborg Universitet* (2026).

[5] Anders Nyby Christensen. *P10 JS Energy*. Accessed: 08-06-2026. 2026. URL: https://github.com/NybyDK/P10-JS-Energy.

[6] Marco Couto, João Saraiva, and João Paulo Fernandes. "Energy Refactorings for Android in the Large and in the Wild". In: *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)* (2020). URL: https://ieeexplore.ieee.org/document/9054858.

[7] João De Macedo et al. "WebAssembly versus JavaScript: Energy and Runtime Performance". In: *2022 International Conference on ICT for Sustainability (ICT4S)*. 2022, pp. 24–34. DOI: 10.1109/ICT4S55073.2022.00014.

[8] Anthony DiBenedetto and Fadi Wedyan. "An Empirical Evaluation of Energy Consumption Across Web Browsers". In: *2025 10th International Conference on Fog and Mobile Edge Computing (FMEC)*. IEEE, 2025. DOI: 10.1109/FMEC65595.2025.11119363.

[9] Facebook. *Hermes JS Engine*. Accessed: 19-05-2026. URL: https://github.com/facebook/hermes.

[10] Facebook Hermes Team. *Hermes Compilation and Runtime Modes*. Accessed: 01-06-2026. URL: https://github.com/facebook/hermes/blob/static_h/doc/blog/2025-11-02-hermes-compilation-runtime-modes.md.

[11] Firefox. *SpiderMonkey*. Accessed: 01-06-2026. URL: https://firefox-source-docs.mozilla.org/js/index.html.

[12] Forsyningstilsynet. *The Danish Electricity and Natural Gas Markets 2022*. Accessed: 01-06-2026. 2023. URL: https://forsyningstilsynet.dk/Media/638282924096043761/The%20Danish%20Electricity%20and%20Natural%20Gas%20Markets%202022.pdf.

[13] Green Software Foundation. *Measure What Matters: Software's Carbon Impact*. Accessed: 08-06-2026. URL: http://greensoftware.foundation/standards/sci/.

[14] Green Software Foundation. *Standards*. Accessed: 01-06-2026. URL: https://greensoftware.foundation/standards/#standards.

[15] OpenJS Foundation. *ESLint*. Accessed: 01-06-2026. URL: https://www.npmjs.com/package/eslint.

[16] OpenJS Foundation. *Find and fix problems in your JavaScript code*. Accessed: 08-06-2026. URL: https://eslint.org/.

[17] Andreas Gal et al. "Trace-based Just-In-Time Type Specialization for Dynamic Languages". In: *SIGPLAN Notices* 44.6 (2009), pp. 465–478. DOI: 10.1145/1543135.1542528. URL: https://doi.org/10.1145/1543135.1542528.

[18] Andy Georges, Dries Buytaert, and Lieven Eeckhout. "Statistically Rigorous Java Performance Evaluation". In: *Proceedings of the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications (OOPSLA '07)*. 2007, pp. 57–76. DOI: 10.1145/1297027.1297033. URL: https://doi.org/10.1145/1297027.1297033.

[19] Max van Hasselt et al. "Comparing the Energy Efficiency of WebAssembly and JavaScript in Web Applications on Android Mobile Devices". In: *Proceedings of the 26th International Conference on Evaluation and Assessment in Software Engineering (EASE 2022)*. Association for Computing Machinery, 2022, pp. 140–149. DOI: 10.1145/3530019.3530034.

[20] Apple Inc. *JavaScriptCore - WebKit Documentation*. Accessed: 19-05-2026. URL: https://docs.webkit.org/Deep%20Dive/JSC/JavaScriptCore.html.

[21] Apple Inc. *WebKit Overview*. Accessed: 01-06-2026. URL: https://docs.webkit.org/.

[22] Deno Land Inc. *A Brief History of JavaScript*. Accessed: 01-06-2026. URL: https://deno.com/blog/history-of-javascript.

[23] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual*. Volume 3: System Programming Guide, Chapter 16: Power and Thermal Management, Section 16.10 (RAPL), Accessed: 19-11-2025. 2025. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

[24] ECMA International. *ECMAScript Language Specification*. 16th. Accessed: 19-05-2026. 2025. URL: https://ecma-international.org/publications-and-standards/standards/ecma-262/.

[25] JetBrains. *State of Developer Ecosystem Report 2024*. Accessed: 01-06-2026. 2024. URL: https://www.jetbrains.com/lp/devecosystem-2024/.

[26] jsPerf. *online JavaScript performance benchmark*. Accessed: 19-05-2026. URL: https://jsperf.app/.

[27] Nicolas van Kempen et al. “It’s Not Easy Being Green: On the Energy Efficiency of Programming Languages”. In: *arXiv preprint arXiv:2410.05460* (2024). Accessed: 19-05-2026. URL: https://arxiv.org/abs/2410.05460.

[28] Kashif Nizam Khan et al. “RAPL in Action: Experiences in Using RAPL for Power Measurements”. In: *ACM Transactions on Modeling and Performance Evaluation of Computing Systems* 3.2 (2018), pp. 1–26. DOI: 10.1145/3177754.

[29] Ivan Krasilnikov. *JavaScript engines zoo*. Accessed: 19-05-2026. URL: https://zoo.js.org/.

[30] Mohit Kumar, Youhuizi Li, and Weisong Shi. “Energy Consumption in Java: An Early Experience”. In: *IGSC* (2017). DOI: 10.1109/IGCC.2017.8323579. URL: https://weisongshi.org/papers/kumar17-eejava.pdf.

[31] Google Chrome Labs. *JavaScript (engine) Version Updater*. Accessed: 19-05-2026. 2026. URL: https://github.com/GoogleChromeLabs/jsvu.

[32] Mozilla MDN. *JavaScript*. Accessed: 01-06-2026. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript.

[33] Mozilla MDN. *JavaScript data types and data structures*. Accessed: 01-06-2026. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Data_structures.

[34] Mozilla MDN. *Just-In-Time Compilation (JIT)*. Accessed: 01-06-2026. URL: https://developer.mozilla.org/en-US/docs/Glossary/Just_In_Time_Compilation.

[35] MeasureThat.net. *Find out how to improve the speed of your web application*. Accessed: 19-05-2026. URL: https://www.measurethat.net/.

[36] Darius Mercadier. *Land ahoy: leaving the Sea of Nodes*. Accessed: 19-05-2026. URL: https://v8.dev/blog/leaving-the-sea-of-nodes.

[37] Juan Merelo Guervós, Mario Garcia-Valdez, and Pedro Castillo. “An Analysis of Energy Consumption of JavaScript Interpreters with Evolutionary Algorithm Workloads”. In: *Proceedings of the 18th International Conference on Software Technologies (ICSOFT 2023)*. 2023, pp. 175–184. DOI: 10.5220/0012128100003538.

[38] Inc. Moddable Tech. *Empower your IoT vision with Moddable*. Accessed: 19-05-2026. URL: https://www.moddable.com/.

[39] Mozilla. *Components of SpiderMonkey*. Accessed: 19-05-2026. URL: https://firefox-source-docs.mozilla.org/js/index.html#components-of-spidermonkey.

[40] Mozilla. *SpiderMonkey*. Accessed: 19-05-2026. URL: https://spidermonkey.dev/.

[41] Oracle. *GraalJS*. Accessed: 19-05-2026. URL: https://www.graalvm.org/javascript/.

[42] Oracle. *JMH Blackhole*. Accessed: 01-06-2026. URL: https://javadoc.io/doc/org.openjdk.jmh/jmh-core/1.23/org/openjdk/jmh/infra/Blackhole.html.

[43] Stack Overflow. *Stack Overflow Developer Survey 2025*. Accessed: 01-06-2026. 2025. URL: https://survey.stackoverflow.co/2025/technology#1-programming-scripting-and-markup-languages.

[44] Taemin Park et al. "NoJITsu: Locking Down JavaScript Engines". In: *Proceedings 2020 Network and Distributed System Security Symposium* (2020). URL: https://api.semanticscholar.org/CorpusID:211269239.

[45] Rui Pereira et al. "Ranking programming languages by energy efficiency". In: *Science of Computer Programming* 205 (2021), p. 102609. ISSN: 0167-6423. DOI: https://doi.org/10.1016/j.scico.2021.102609. URL: https://www.sciencedirect.com/science/article/pii/S0167642321000022.

[46] Dmytro Piddubnyi. *Benchmark JavaScript and TypeScript Instantly*. Accessed: 19-05-2026. URL: https://www.jsperf.net/.

[47] Dennis Pockstaller, Stefan Huber, and Lukas Demetz. "Comparing the Energy Consumption of WebAssembly and JavaScript in Mobile Browsers". In: *Proceedings of the 19th International Conference on Web Information Systems and Technologies (WEBIST 2023)*. SCITEPRESS – Science and Technology Publications, Lda, 2023, pp. 121–127. DOI: 10.5220/0012205600003584.

[48] Chapel Project. *The Chapel Parallel Programming Language*. Accessed: 19-05-2026. URL: https://chapel-lang.org/.

[49] QuickJS-NG Project. *Welcome to QuickJS-NG*. Accessed: 19-05-2026. URL: https://quickjs-ng.github.io/quickjs/.

[50] V8 Project. *V8 is Faster and Safer than Ever!* Accessed: 19-05-2026. 2023. URL: https://v8.dev/blog/holiday-season-2023.

[51] V8 Project. *What is V8?* Accessed: 19-05-2026. URL: https://v8.dev/.

[52] Stephen Röttger. *Control-flow Integrity in V8*. Accessed: 19-05-2026. URL: https://v8.dev/blog/control-flow-integrity.

[53] Marija Selakovic and Michael Pradel. "Performance Issues and Optimizations in JavaScript: An Empirical Study". In: *Proceedings of the 38th International Conference on Software Engineering (ICSE 2016)*. 2016, pp. 61–72. DOI: 10.1145/2884781.2884829.

[54] The World Bank and International Telecommunication Union. *Measuring the Emissions & Energy Footprint of the ICT Sector: Implications for Climate Action*. Tech. rep. ISBN 978-92-61-38541-5. The World Bank and International Telecommunication Union, 2024.

[55] V8. *Blazingly fast parsing, part 2: lazy parsing*. Accessed: 01-06-2026. URL: https://v8.dev/blog/preparser.

[56] V8. *Documentation*. Accessed: 01-06-2026. URL: https://v8.dev/docs.

[57] Roman Zhuykov et al. "Augmenting JavaScript JIT with Ahead-of-Time Compilation". In: *2015 Computer Science and Information Technologies (CSIT)*. Accessed: 01-06-2026. 2015. DOI: 10.1109/CSITechnol.2015.7358262. URL: https://csit.am/2015/proceedings/PDC/PDC7.pdf.

APPENDIX

APPENDIX A: ENGINE WORST-CASES

Worst-case relative energy consumptions for each engine.

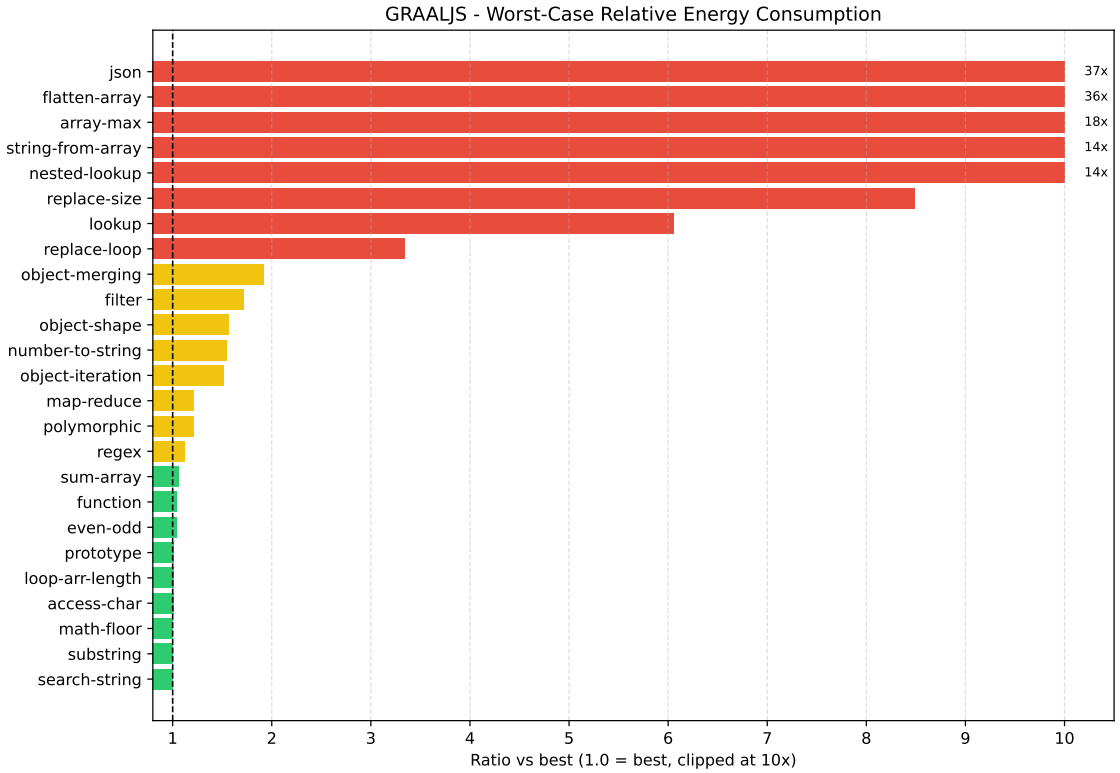


Fig. 12: Worst-case relative energy consumption for every category for the GraalJS engine.

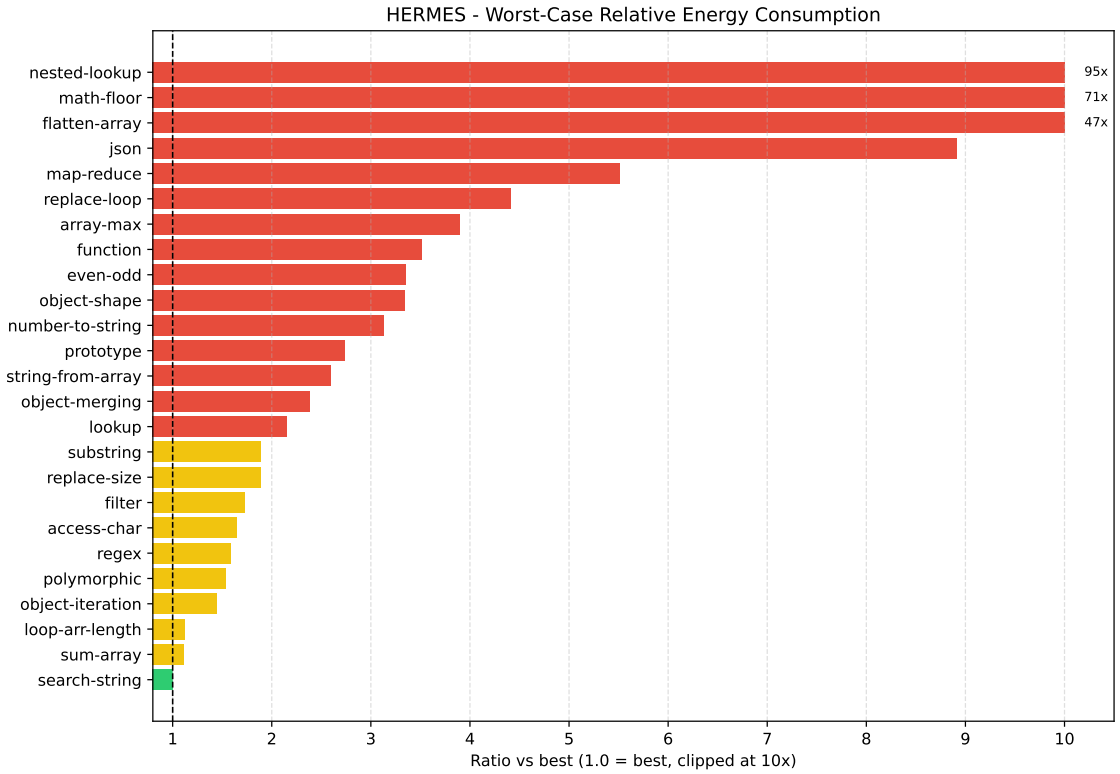


Fig. 13: Worst-case relative energy consumption for every category for the Hermes engine.

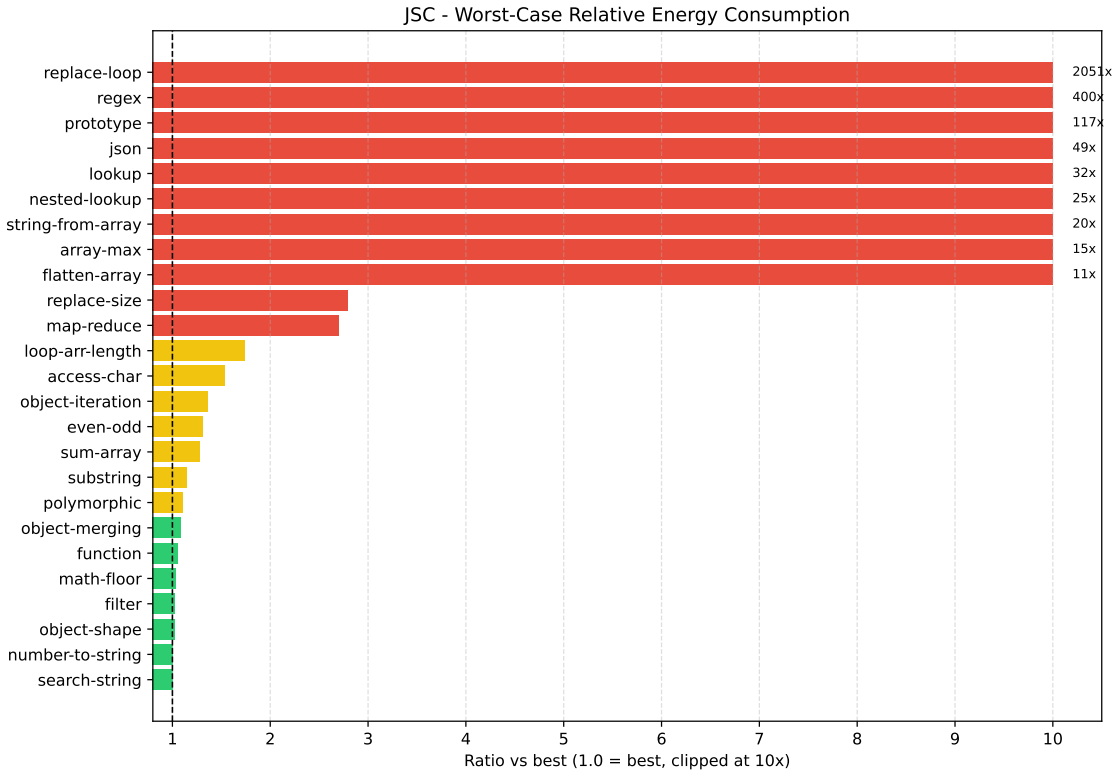


Fig. 14: Worst-case relative energy consumption for every category for the JavaScriptCore engine.

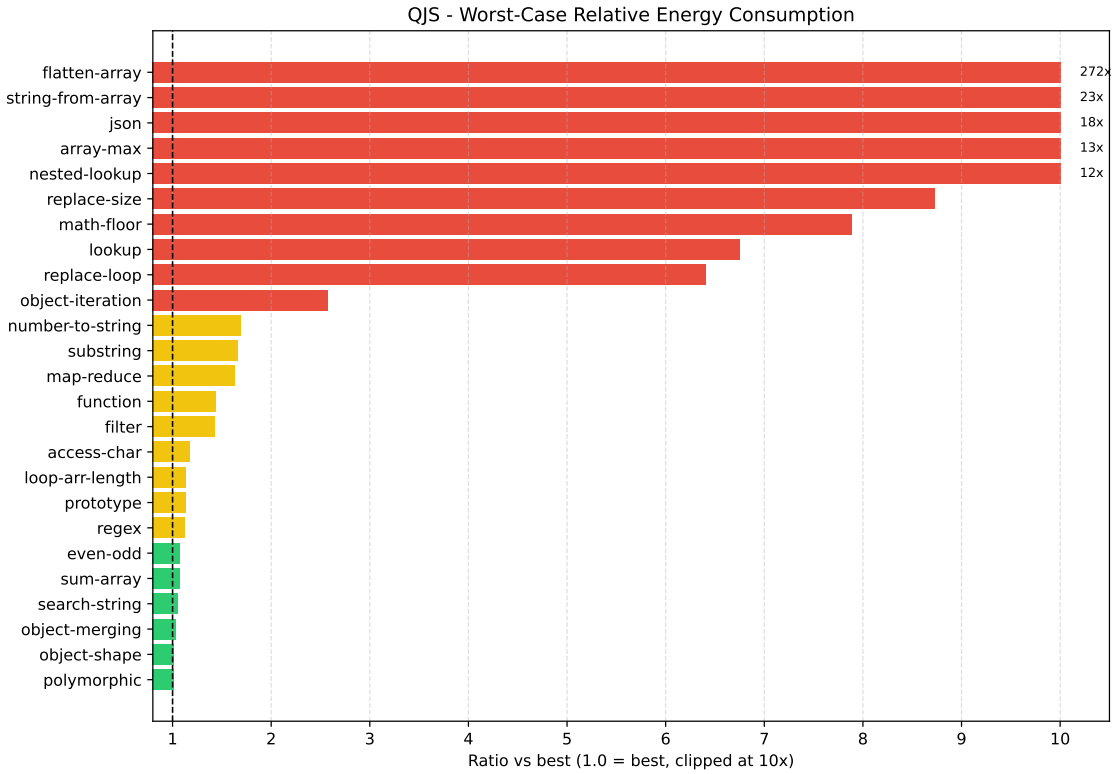


Fig. 15: Worst-case relative energy consumption for every category for the QuickJS engine.

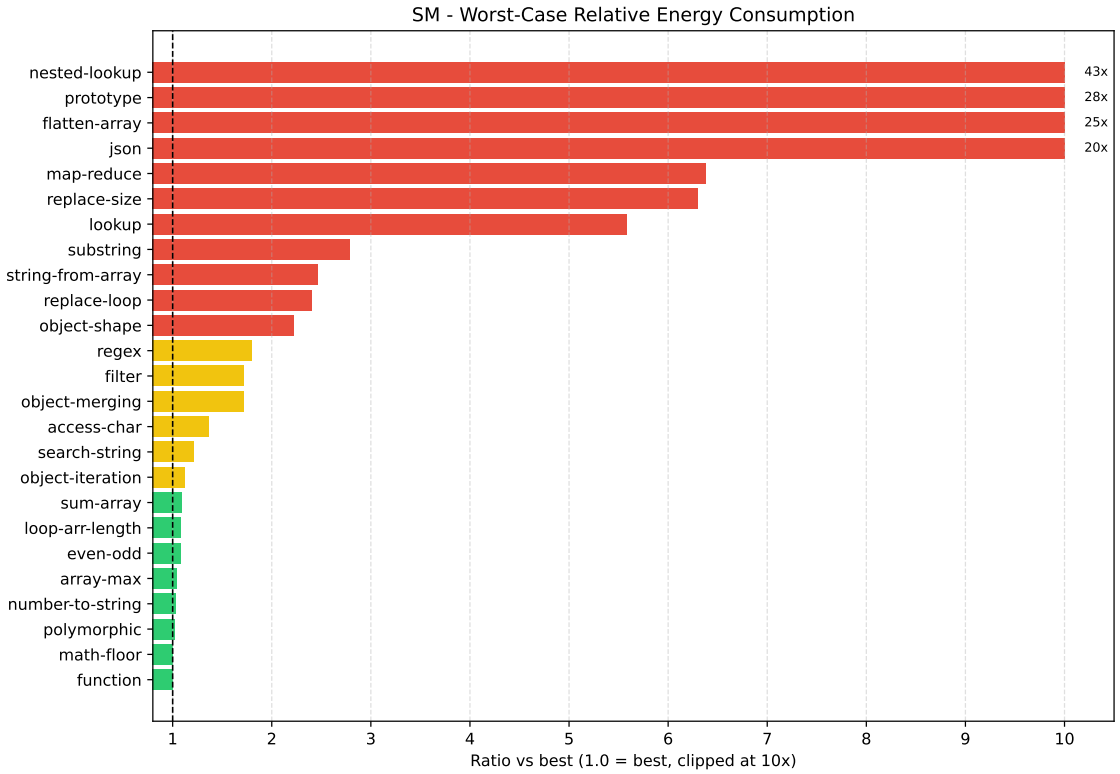


Fig. 16: Worst-case relative energy consumption for every category for the SpiderMonkey engine.

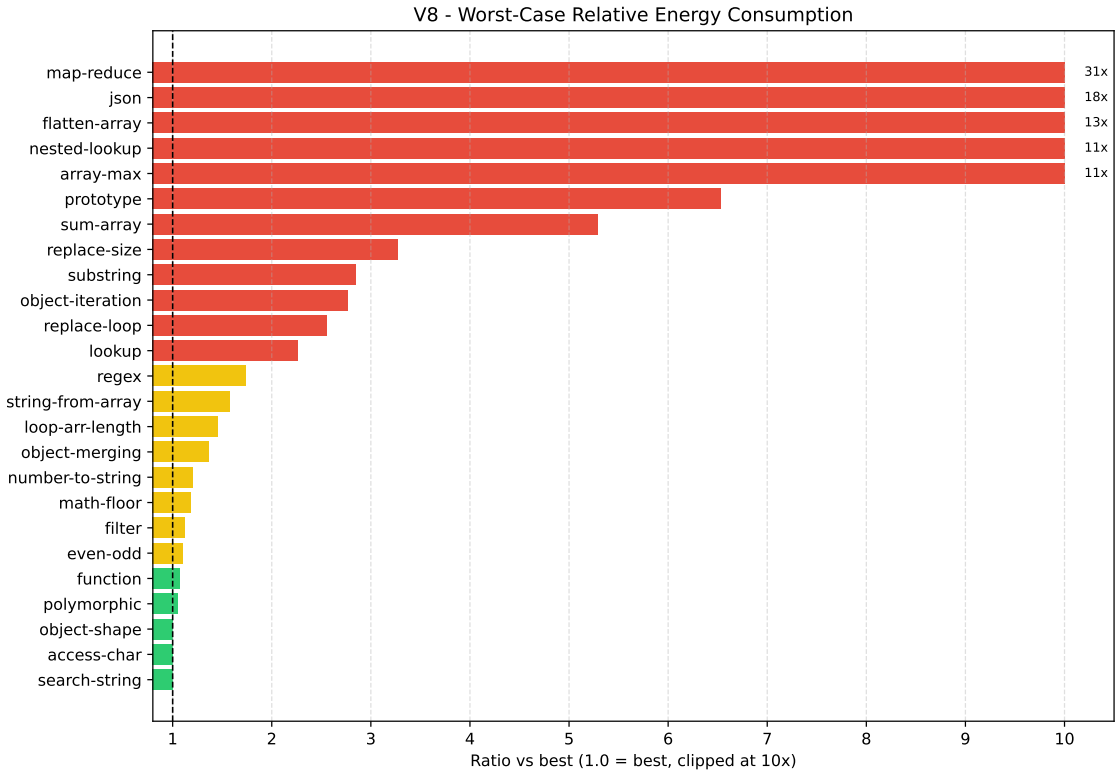


Fig. 17: Worst-case relative energy consumption for every category for the V8 engine.

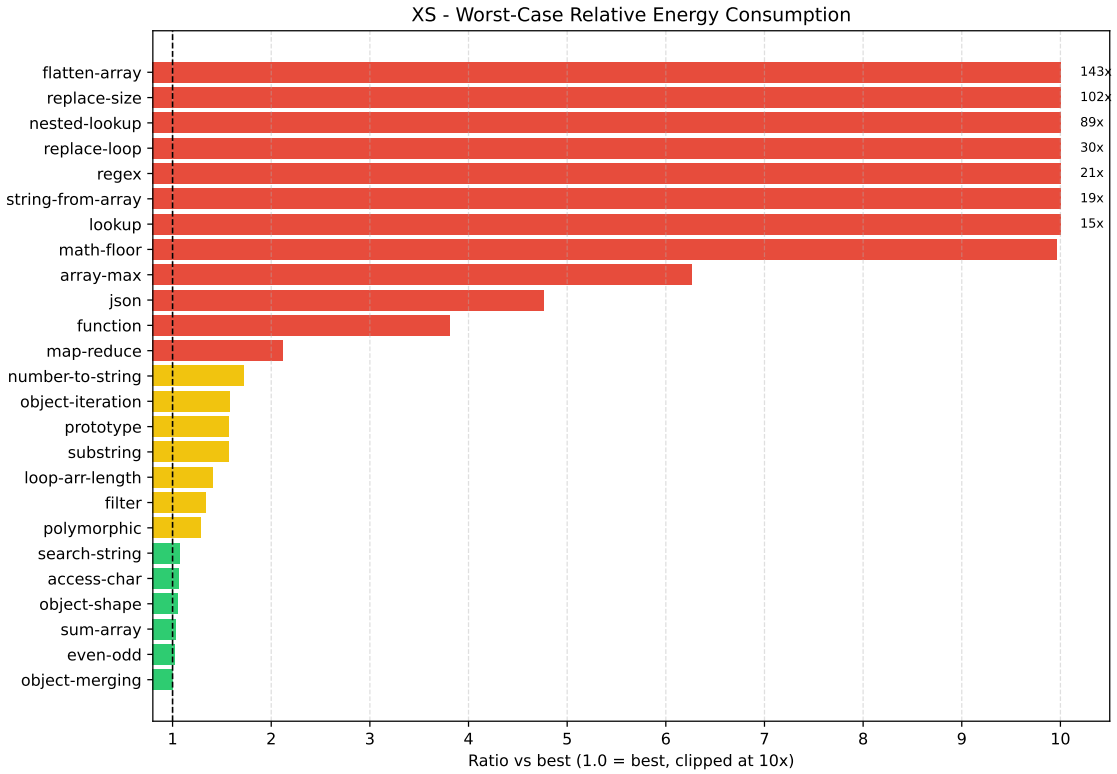


Fig. 18: Worst-case relative energy consumption for every category for the XS engine.

APPENDIX B: ESLINT RULES

ESLint rules for V8.

Listing 24: ESLint rule for flattening an array in V8.

```
export default {
    meta: {
        type: "suggestion",
        docs: {
            description: "Prefer [].concat(...arr) for flattening arrays",
        },
        fixable: "code",
        schema: [],
    },

    create(context) {
        return {
            CallExpression(node) {
                // arr.flat()
                if (node.callee.type === "MemberExpression" &&
                    ↪ node.callee.property.name === "flat") {
                    context.report({
                        node,
                        message: "Use [].concat(...arr) instead of flat() for up to
                            ↪ 1.46x better performance",
                        fix(fixer) {
                            const sourceCode = context.sourceCode;
                            const obj = sourceCode.getText(node.callee.object);

                            return fixer.replaceText(node, `[].concat(...${obj})`);
                        },
                    });
                }

                // arr.reduce(concat)
                if (node.callee.type === "MemberExpression" &&
                    ↪ node.callee.property.name === "reduce") {
                    const [callback] = node.arguments;

                    if (callback && callback.type === "ArrowFunctionExpression") {
                        const body = callback.body;

                        if (
                            body.type === "CallExpression" &&
                            body.callee.type === "MemberExpression" &&
                            body.callee.property.name === "concat"
                        ) {
                            context.report({
                                node,
```

```
                                message:
                                    "Use [].concat(...arr) instead of reduce+concat
                                        ↪ for up to 13x better performance",
                                fix(fixer) {
                                    const sourceCode = context.sourceCode;
                                    const obj =
                                        ↪ sourceCode.getText(node.callee.object);

                                    return fixer.replaceText(node,
                                        ↪ `[].concat(...${obj})`);
                                },
                            });
                        }
                    }
                }
            },
        };
    },
};
```

Listing 25: ESLint rule for flattening an array in V8.

```
export default {
    meta: {
        type: "suggestion",
        docs: {
            description: "Prefer loop-based max instead of Math.max()",
        },
    },

    create(context) {
        return {
            CallExpression(node) {
                // Math.max(...arr)
                const isMathMax =
                    node.callee.type === "MemberExpression" &&
                    node.callee.object.name === "Math" &&
                    node.callee.property.name === "max";

                const hasSpread = node.arguments && node.arguments.some(arg =>
                    ↪ arg.type === "SpreadElement");

                if (isMathMax && hasSpread) {
                    context.report({
                        node,
                        message: "Avoid Math.max(...arr), use loop-based max.",
                    });
                }

```

```
                // Math.max.apply(null, arr)
                if (
                    node.callee.type === "MemberExpression" &&
                    node.callee.object.type === "MemberExpression" &&
                    node.callee.object.object.name === "Math" &&
                    node.callee.object.property.name === "max" &&
                    node.callee.property.name === "apply"
                ) {
                    const args = node.arguments;

                    if (args.length === 2 && args[0]?.type === "Literal" && args[1]) {
                        context.report({
                            node,
                            message: "Avoid Math.max.apply(null, arr), use loop-based
                                ↪ max.",
                        });
                    }
                }
            },
        };
    },
};
```

Listing 26: ESLint rule for flattening an array in V8.

```
export default {
    meta: {
        type: "suggestion",
        docs: {
            description: "Avoid JSON.parse inside loops due to energy overhead",
        },
    },

    create(context) {
        function isJSONParse(node) {
            return (
                node.type === "CallExpression" &&
                node.callee.type === "MemberExpression" &&
                node.callee.object.name === "JSON" &&
                node.callee.property.name === "parse"
            );
        }

        function checkLoop(node) {
            function traverse(n) {
                if (!n) return false;
                if (isJSONParse(n)) return true;

                for (const key in n) {
```

```
                    const value = n[key];
                    if (value && typeof value === "object") {
                        if (traverse(value)) return true;
                    }
                }

                return false;
            }

            if (traverse(node.body)) {
                context.report({
                    node,
                    message: "Avoid JSON.parse inside loops (high energy cost observed).",
                });
            }
        }

        return {
            ForStatement: checkLoop,
            WhileStatement: checkLoop,
            ForOfStatement: checkLoop,
            ForInStatement: checkLoop,
        };
    },
};
```

Listing 27: ESLint rule for flattening an array in V8.

```
export default {
    meta: {
        type: "suggestion",
        docs: {
            description: "Prefer Math.floor().",
        },
    },

    create(context) {
        return {
            // num | 0
            BinaryExpression(node) {
                if (node.operator === "|") {
                    const isZero = node.right.type === "Literal" && node.right.value === 0;

                    if (isZero) {
                        context.report({
                            node,
                            message: "Avoid num | 0. Use Math.floor(num).",
```

```
                        });
                    }
                }
            },

            // ~~num
            UnaryExpression(node) {
                if (node.operator === "~") {
                    if (node.argument.type === "UnaryExpression" &&
                        ↪ node.argument.operator === "~") {
                        context.report({
                            node,
                            message: "Avoid ~~num. Use Math.floor(num).",
                        });
                    }
                }
            },

            // parseInt(num)
            CallExpression(node) {
                if (node.callee.type === "Identifier" && node.callee.name ===
                    ↪ "parseInt") {
                    context.report({
                        node,
                        message: "Avoid parseInt(num). Use Math.floor(num).",
                    });
                }
            },
        };
    },
};
```