Investigating Frequency Scaling and Scheduling for JavaScript Web API Optimization

Daniel HyunSeok Jang
Computer Science, NYUAD
hsj276@nyu.edu

Advised by: Yasir Zaki, Matteo Varvello

ABSTRACT

Increasing number of smartphones have been equipped with the heterogeneous multicore system, which consists of dissimilar processors to balance performance and energy efficiency of the device. To efficiently utilize this CPU architecture for mobile web browsing, considerable optimization solutions have implemented Application-Assisted Scheduling (AAS) and frequency scaling. This paper evaluates whether this technique could be applied to optimize the performance-energy tradeoff at a finer granularity of JavaScript (JS) Web APIs. To this end, we report on our benchmarks of JS API executions at different CPU configurations. We found that executing APIs at "big" cores running at high clock speed generally results in the best balance between duration and battery consumption, though each API's optimal CPU configuration may vary depending on execution-specific factors. The comparison of optimal CPU configurations with respect to the default Linux governors show that, when frequency scaling and scheduling are done judiciously, each API execution could benefit from a 26-41% reduction in energy consumption, a 4-27% reduction in duration, and a 9-46.2% reduction in Energy Delay Product (EDP). Assessing the impact of JS API level optimization to the overall user experience is our future work.

KEYWORDS

Mobile Web Browsing, CPU Configuration, Energy Optimization

This report is submitted to NYUAD's capstone repository in fulfillment of NYUAD's Computer Science major graduation requirements.

جامعة نيويورك أبوظبي
NYU ABU DHABI

Capstone Project 2, Spring 2023, Abu Dhabi, UAE

Reference Format:
Daniel HyunSeok Jang. 2023. Investigating Frequency Scaling and Scheduling for JavaScript Web API Optimization. In *NYUAD Capstone Project 2 Reports, Spring 2023, Abu Dhabi, UAE.* 8 pages.

1 INTRODUCTION

Web browsing is one of the most popular smartphone activities, as approximately 60% of universal web traffic accounted for mobile devices in 2022. An open problem in mobile web browsing is balancing the performance-energy tradeoff. Users expect smooth responsiveness while interacting with the web browser, but such a high level of performance necessitates heavy CPU usage, potentially causing significant battery drainage of the device. This problem often stems from poor utilization of the device's heterogeneous multicore systems, which requires the OS kernel to accurately determine the complexity of every task to make effective scheduling decisions. Unfortunately, Android devices have not made optimal use of this architecture for handling web browsing activities, primarily because the operating system kernel lacks the knowledge of webpage-specific workload and is prone to making suboptimal scheduling decisions [3, 13–15, 17].

A common approach to this problem is frequency scaling and Application-Assisted Scheduling (AAS). This mechanism allows for the browser to participate in scheduling and decide which CPU configuration (type of CPU and frequency) to use to handle specific web browsing tasks. Our research is motivated by the fact that its potential optimization of JavaScript (JS) execution has been relatively unexplored. In particular, we focus on the set of JS Web APIs from [5], which is a representative set of APIs commonly executed by the browser's JS engine throughout web browsing activities. To evaluate the degree of optimization through frequency scaling and scheduling at JS API level, we benchmarked the performance and energy tradeoff from executing APIs at different CPU configurations, and discuss our findings in this paper.

2 BACKGROUND

2.1 Scheduling in Heterogeneous Multicore Systems

Heterogeneous multicore systems are composed of different types of cores which are specialized for processing particular tasks. In mobile SoCs, this CPU topology is commonly used to achieve better energy efficiency. Most notably, ARM's big.LITTLE architecture pairs high-performance and high power "big" cores with energy-efficient yet less performant "LITTLE" cores, allowing the scheduler to allocate high-intensity jobs on big cores and lightweight jobs on little cores to improve battery life with minimal performance tradeoff. For example, XiaoMi's Redmi Note 10 5G, the device used throughout our benchmarks, comes with 2 Cortex-A76 (big) cores and 6 Cortex-A55 (LITTLE) cores. These two core clusters operate on different ranges of frequencies, as the big cores scale from 0.725GHz to 2.203GHz while the LITTLE cores scale from 0.5Ghz to 2GHz. Hence, the performance and energy consumption of running any task on heterogeneous-multicore mobile devices significantly depend on the allocated CPU cluster (big or LITTLE) and their clock frequencies.

Figure 1 shows the power profile values of all CPU configurations for RedMi Note 10 5G. Provided by the device manufacturer, the values specify the required power consumption (in mA) of running each CPU cluster at a specific frequency [1]. At its highest clock speed of 2GHz, a LITTLE core only consumes a third of power required for a big core at its highest frequency of 2.23GHz (74.45mA vs 212.36mA). In addition, the power profile estimates that running at half of the big core's maximum frequency (1.129GHz) could yield more than 62.3% reduction in energy consumption (79.92mA vs 212.36mA). The examples both demonstrate the potential of enhancing the battery life by leveraging the heterogeneous multicore architecture and scaling frequencies. In Android operating systems, the *CPUfreq governor* is the driver that defines the characteristics of the system cores, such as the rules for changing the cluster frequencies. There are several types of governors, each with its own behavior and purpose. The *powersave* governor intends to maximize energy savings by setting CPUs at their lowest frequency, while the *performance* governor optimizes for heavy workload by setting CPUs at their highest frequency. The default governor in recent Android smartphones is *schedutil* which provides dynamic frequency scaling: adjusting the clock speed with respect to the system workload to balance performance and battery life. Finally, the *userspace* governor allows the CPU system to run at frequencies configured by any processes running as root.

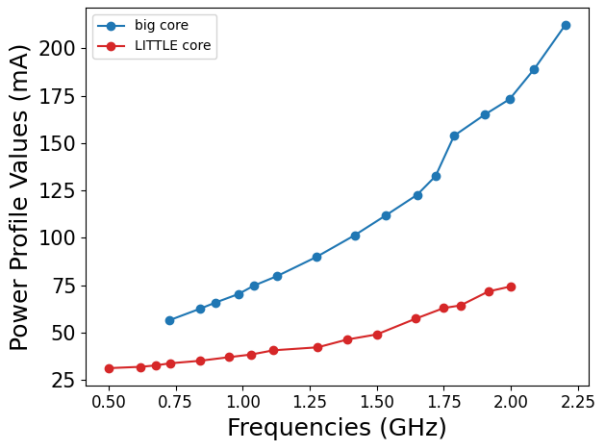


Figure 1: Power Profile Values for CPU Frequencies of RedMi Note 10 5G

2.2 Chromium Architecture

Chromium is an open-source browser that provides the building codebase for many popular web browsers, including Google Chrome, Opera, Microsoft Edge, and Brave. These browsers follow a multi-process architecture in which there is a single *Browser* process and multiple *Renderer* processes. The *Browser* process is responsible for the general UI, network requests, and management of *Renderer* processes. Each *Renderer* process usually corresponds to a browser tab, using the Blink rendering engine and v8 JavaScript engine to handle web browsing activities. The processes communicate with each other over IPC channels and shared memory, where the *Browser* process allocates the fetched web contents into the Shared Resource Buffer for the *Renderer* process to read and parse.

The *Renderer* process itself is further divided into multiple threads, each with a specific role. The most important thread is the *main* thread, which handles the DOM and CSS parsing, JS execution, and the Render steps (style, layout, and paint). Thus each JS engine instance of a *Renderer* process schedules and executes JS in a single-threaded fashion per frame [12]. A developer may implement multithreaded execution of JS by utilizing Web Workers, but they are strictly limited to CPU-bound tasks that do not access the DOM in order to ensure thread-safety [7]. Other threads include the compositor thread, which composites the layers of web content into the final image displayed on screen by generating GPU commands.

3 RELATED WORK

Web browsing activity can be broadly classified into 2 phases: load and interaction. This section provides an overview of prior optimization strategies proposed in the literature to

enhance the performance and energy efficiency of mobile devices for each of these phases

3.1 Frequency Scaling and AAS for Energy Optimization

Several papers have explored and implemented the potential benefits of dynamic voltage and frequency scaling (DVFS) along with Application-Assisted Scheduling (AAS) for mobile web browsing. Their common goal was to predict the workload of web browsing activities, and determine their optimal CPU configurations. Some notable examples utilized machine-learning techniques and trained on static components of websites (such as HTML, CSS and DOM information) to develop prediction models of webpage-specific workload, which enabled energy savings during the loading phase [13, 14, 18]. [3] proposed to integrate Quality of Service into AAS, in which the browser's threads would only migrate to the big cluster when the frame rate gets below a specified threshold.

To reduce battery consumption during webpage interactions, [19] adjusted the CPU configuration throughout runtime based on the expected latency and energy consumption of frames triggered by browser events. [15] and [17] both developed artificial neural networks to predict the minimum frame latency for an user interaction and determine the optimal core configuration.

Our work is motivated by the findings of these prior studies that have successfully achieved battery savings during mobile web activities, without significantly compromising performance, through frequency scaling and scheduling. Given the increasing complexity of JavaScript in modern web applications, we investigated whether a similar optimization approach could be applied to the finer level of granularity offered by JS Web APIs. Therefore, our benchmarks serve as a complement to previous research in this area.

3.2 JavaScript Analysis

Previous studies at JavaScript analysis largely focused on minimizing the initial PLT (page load time) through code mitigation. Uglifiers are widely used by web developers to compress the size of JS files by removing all unnecessary characters without changing functionality [2]. Other solutions include classifying and removing unnecessary JS Web APIs [5], blocking unnecessary JS elements [4], or eliminating unused JS functions of web applications [9]. Polaris improved PLT through optimizing the browser's resource fetch scheduling. In contrast, our work focuses on balancing both performance and energy consumption throughout web browsing activities. A more recent study suggested Web Assembly as the more energy-efficient alternative to JavaScript for web application development [16]. While Web Assembly was beyond the scope of our focus on JavaScript Web APIs, this paper also points at the energy efficiency of JavaScript in modern web applications and possible room for further optimization.

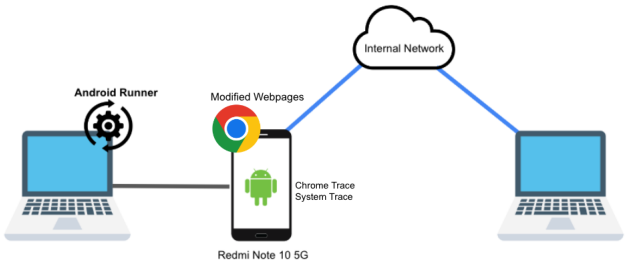


Figure 2: Benchmark Setup

4 METHODOLOGY

The benchmark methodology aims to measure the duration and energy consumption required for executing common JavaScript Web APIs at different CPU configurations. This section overviews the setup and the technical steps involved in our benchmark pipeline.

4.1 Benchmark Setup

As shown in Figure 2, the setup consisted of two desktop computers and an Android smartphone. We used Xiaomi Redmi Note 5G as a representative mobile platform with heterogeneous multicore system. The smartphone was connected via usb (charging was disabled during the benchmark) to the second desktop in order to utilize Android Runner [10], a framework for executing customized interactions and experiments on Android devices. We developed a script to initiate Android Runner for automating web browsing activities at specific CPU configurations and subsequently collect trace files for analysis, which we elaborate further in section 4.2.2.

The second desktop was used as a web server for 20 most popular online news sites, in which their HTML and JS files were modified beforehand for the benchmark. Using Esprima [8], we identified every occurrence of JS Web APIs in the web application codebase and wrapped each with *Performance Timing Markers* [11], as shown in Figure 3. The performance object provides 2 helper functions: *performance.mark* to record named timestamps and *performance.measure* to calculate time between named timestamps. By assigning an unique identifier to the name of each *Performance Timing Marker*, we were able to track the start time and duration of every JS Web API execution at microsecond precision.

Figure 3: Example of inserted performance marks

4.2 Benchmark Pipeline

Each benchmark run evaluated the performance-energy trade-off of executing JS Web APIs at a specified CPU cluster and frequency. To this end, we constructed a pipeline broadly composed of 3 stages:

4.2.1 *Setting Cluster Frequency and Chrome CPU Affinity.*

Before simulating web browsing activities, we configured each CPU cluster to a particular clock speed by leveraging CPU performance scaling supported in LINUX-based systems. We wrote a simple shell script to set the governor of all cores to *userspace* and subsequently modify their frequency values in their respective *scaling_setspeed* files in *sysfs*. After adjusting the processor settings, we altered the CPU affinity of all threads belonging to Google Chrome's Renderer process using the *tasket* system call. The CPU affinity of relevant threads would be set from 0th to 5th cores (6xCortex-A55) to execute web browsing tasks on the LITTLE cluster, and to the 6th and 7th core (2xCortex-A76) to solely utilize the big cluster. Unfortunately, CPU affinity does not guarantee that a thread would always be allocated to its associated core(s), so we disabled the unused cluster for each benchmark to prevent misallocations. On average, 88.2% of slices of the Renderer process' *main* thread were executed appropriately at the specified CPU cluster and frequency. We observed that, even after configuration, the clock speeds of certain cores within a cluster infrequently fluctuated during the benchmarks. We omitted such misallocated slices in our analysis. We ran our benchmarks at 14 different CPU configurations, 7 from each CPU cluster.

4.2.2 *Profiling Webpage Load and Interaction.*

We customized Android Runner to automate web browsing activities. Specifically, for each of the 20 modified news websites, the device would load its web contents from an internal network and perform programmed interactions. The interactions included common browsing gestures, such as scrolling and pinching. We noticed that initially loading a webpage through Android Runner briefly sets the frequencies of all cores to their maximum clock speeds. As a workaround, our interaction script cleared the browser cache and reloaded each page, which did not cause such sudden spikes in CPU frequencies. For robustness, we excluded any data collected before the reload from our analysis.

Throughout each webpage activity, both *Chrome Tracing* and Android's *System Tracing* were conducted in the background. *Chrome Tracing* profiles threads and activities spawned by the Chrome browser, including the inserted performance marks for tracking JS Web API executions. *System Tracing* uses Linux *ftrace* under the hood to report the allocated thread and frequency of the device's CPUs. The 2 generated trace files were collected after every webpage benchmark for analysis.

4.2.3 *Profile Analysis and Energy Consumption Evaluation.*

Perfetto's Python API was utilized to process and analyze JS Web API executions collected from the benchmarks. Our analysis is based on the fact that JS is executed in a single-threaded fashion by the *Renderer* process' *main* thread. Hence, the CPU configuration(s) for a particular JS Web API execution must correspond to that of the main thread during the execution time interval. By synchronizing the timestamps of API executions recorded in the *Chrome Tracing* file with the main thread's processor allocations recorded in the *System Tracing* file, we were able to identify the cores and their frequencies involved in each execution. Note that each API execution could be split into multiple slices due to context switchings, and each slice could be allocated to a different core (potentially running at a different frequency). In other words, API execution and CPU configuration is modeled by a one-to-many relationship, while slice and CPU configuration must be one-to-one.

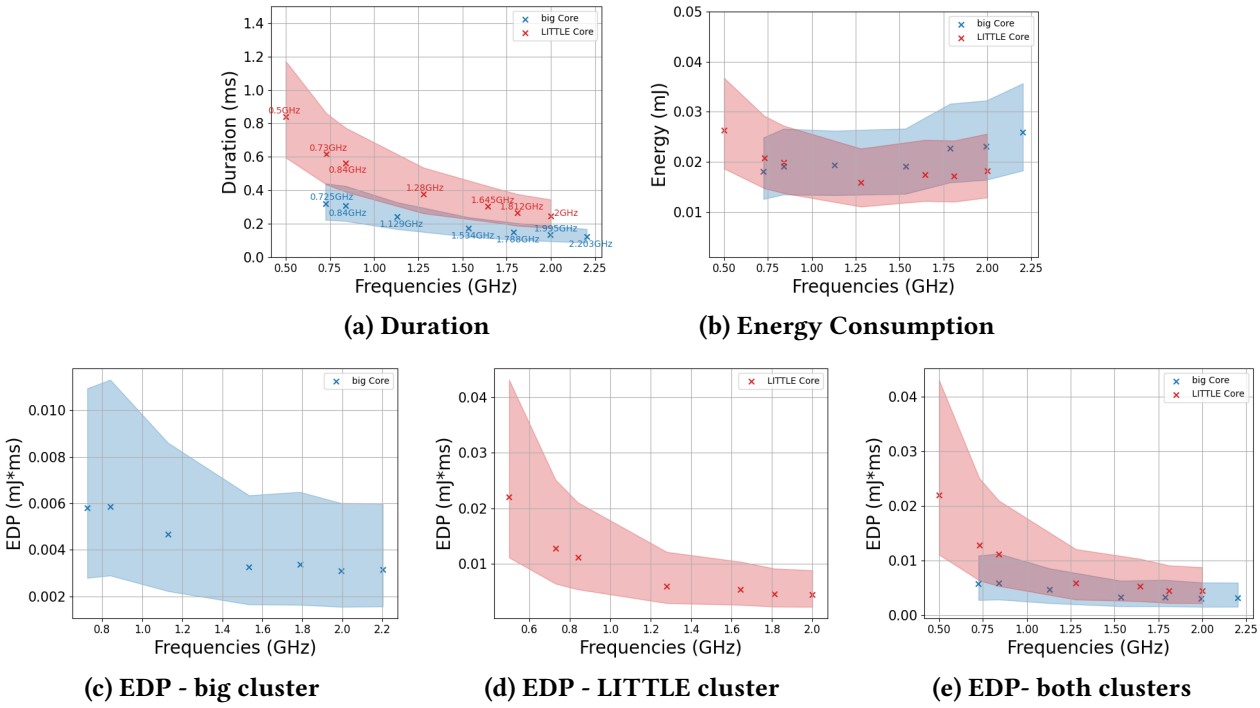
Let F = $\{f_1, f_2 ... f_n\}$ be a set of JS Web API executions, where f_i consists of a set of slices $S_i = \{s_1, s_2 ... s_m\}$. We can calculate the energy consumption of any arbitrary slice s_j by

$$E(s_j) = t_j * P(c_j) \tag{1}$$

where t_j is the duration of s_j, c_j is the CPU configuration of s_j, and $P(c_j)$ denotes the power profile value at c_j (plotted in Figure 1). Then the energy consumption of f_i can be computed as

$$E(f_i) = \Sigma_{s_j \in S_i} E(s_j) \tag{2}$$

In other words, the energy consumption of an API execution is the sum of energy consumption of its slices, each of which is computed as the product of slice duration and the power profile value of the slice's CPU configuration. This approach establishes a comparison of energy consumption required for web browsing activities at different CPU configurations, and is akin to that employed in popular battery profiling tools such as BatteryStats [6].



(a) Duration (b) Energy Consumption

(c) EDP - big cluster (d) EDP - LITTLE cluster (e) EDP- both clusters

Figure 4: Comparison of Optimization Metrics Among Benchmarked CPU Configs.

5 RESULTS

Our benchmarks collected over 9,000,000 different executions across 416 different JS Web APIs. We considered 3 *lower is better* metrics: execution duration, energy consumption, and energy delay product - computed as the product of execution time and energy consumption.

5.1 Duration, Energy, and EDP

We present the comparison of the 3 key metrics across different CPU configurations at Figure 4, where each "x" mark denotes the median value at a benchmarked CPU configuration. We also illustrated the 25th and 75th percentile bound.

The results depicted in Figure 4a are consistent with our expectations, as executing APIs at higher frequencies and at the big cluster resulted in a substantial reduction in execution time. Specifically, the average execution time for APIs at the maximum frequency of the big cluster was found to be 2x faster than that of the LITTLE cluster, while at the minimum frequency, it was 2.6x faster. The LITTLE cluster's maximum frequency outperformed the big cluster only at its two lowest frequencies.

Figure 4b presents a comparison of the average energy consumption during the execution of APIs across the benchmarked core configurations. For the LITTLE cores, we observed that the reduced execution time at higher frequencies initially leads to a lower battery consumption, which persists until 1.28GHz, beyond which the energy consumption scales up with the frequency once again. Similarly, the energy consumption of the big cores is generally aligned with higher frequency. Upon comparing the LITTLE and big clusters, we observed that the big cluster consumes more battery life, especially at high frequencies. Our results indicate that migrating to the LITTLE cluster when both clusters are operating at their maximum clock speed could potentially result in up to 42% energy savings. However, it is worth noting that while the difference in energy efficiency between the two clusters is not negligible, it is comparatively smaller than that observed in the case of execution time (Figure 4a), as revealed by their interquartile ranges.

Figures 4c and 4d depict the average EDP values for the big cluster and LITTLE cluster, respectively. We observed a gradual decrease in EDP with an increase in frequency. Specifically, at its maximum clock speed, the big cluster demonstrated a 1.836 times reduction in EDP when compared to its minimum clock speed. For the LITTLE cluster, the maximum frequency saw 4.9 times reduction in EDP with respect to its minimum. Figure 4e provides a comparison of the EDP values across both clusters. While the API executions at the big cores generally yielded lower EDP, the large variance observed in both clusters, as evidenced by their interquartile range, suggests that the EDP values vary

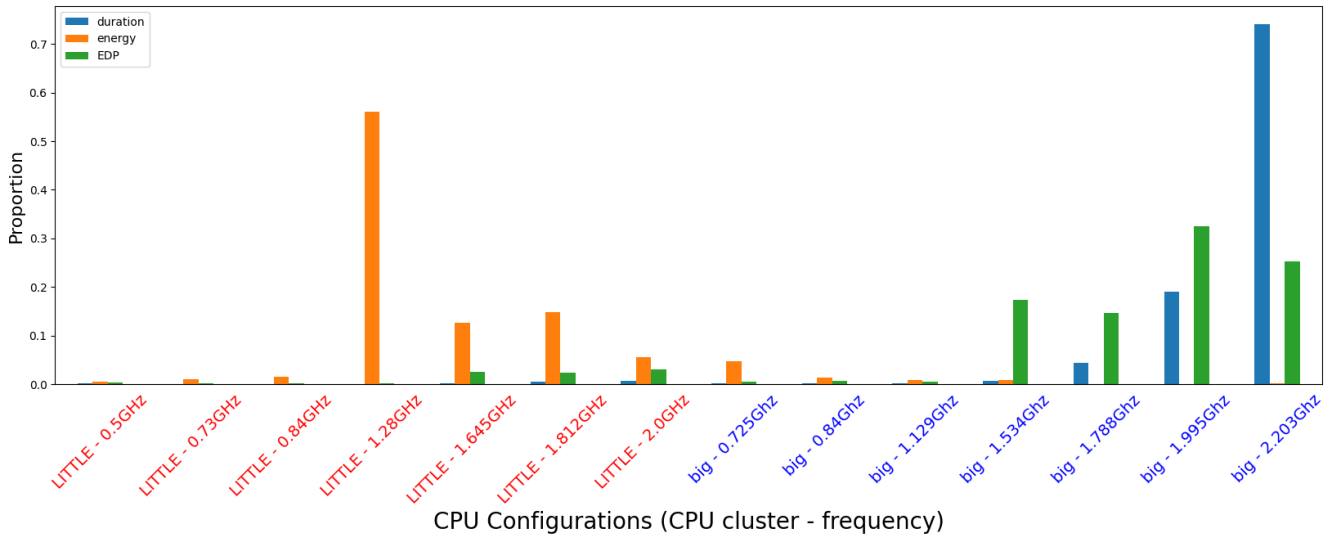


Figure 5: Dist. of Optimal Config. for each Optimization Metric

significantly with each execution, indicating that there may not be a one-size-fits-all configuration for EDP optimization.

5.2 Distribution of Optimal CPU Configs.

We defined "optimal" core configurations for each API execution as those result in the minimum value in the 3 key metrics. Figure 5 depicts the distribution of optimal configurations among our benchmarked core configurations. In terms of performance (measured by duration), the three highest frequencies of the big core were optimal in more than 97% of the APIs executions we benchmarked, of which 76% were at the maximum frequency. In contrast, over 92% of the optimal configurations for energy consumption belonged to the LITTLE cluster. Notably, over 56% of API executions had minimal energy consumption at the median configuration of 1.28GHz, while only 3% were optimal at lower frequencies. This finding indicates that prolonging task executions at low frequencies could potentially harm battery life, even at the level of JS Web APIs.

Regarding the optimal configuration for EDP, we observed that it generally belonged to the big cluster. Specifically, the four highest frequencies of the big cluster were optimal for EDP in over 87% of API executions. In contrast to the distribution of optimal configurations for performance, however, the distribution of optimal configurations for EDP did not scale linearly with frequency. In addition, we noted that the LITTLE cluster at high frequencies optimized EDP by an order of magnitude more than the big cluster at low frequencies. This suggests that certain APIs may be better executed on LITTLE cores, and that the EDP values from executing

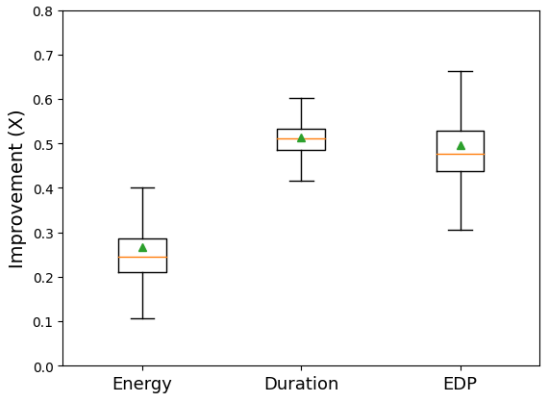


Figure 6: Optimization Metrics of Optimal Config. w.r.t. Benchmarked Config.

at LITTLE cores with high frequencies may not be vastly different from those executed on big cores.

5.3 Expected Improvement from Optimal CPU Configs.

Figure 6 presents the expected improvements in the three key metrics achieved by judiciously scheduling every API execution to its respective optimal CPU configuration. We calculate the expected improvement as the median of improvements from running at the optimal configuration with respect to all other benchmarked CPU configurations. Each boxplot in Figure 5 depicts the interquartile range of all expected improvements, with a green mark and an orange line representing the mean and median, respectively. Our

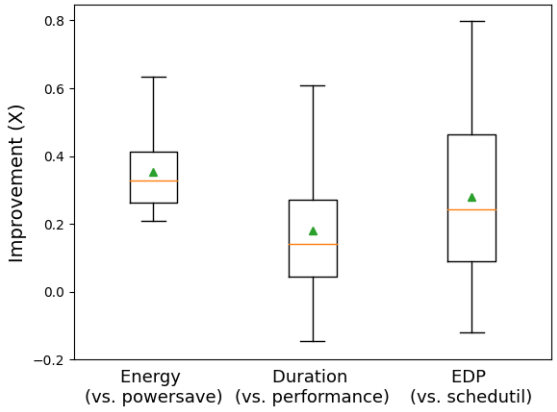


Figure 7: Optimization Metrics of Optimal Config. w.r.t. Linux Governors

analysis shows that leveraging optimal configurations could potentially reduce energy consumption, execution duration, and EDP by 21-28%, 48-53%, and 43-52%, respectively. These results further underscore the benefits of leveraging the optimal configurations to execute JS Web APIs.

5.4 Optimal Configs. vs Linux Governors

We expanded our investigation of optimal configurations through comparisons with 3 different Linux governors - *powersave*, *performance*, and *schedutil* -, as shown in Figure 7. Compared to the core configurations set by the *powersave* governor, the optimal configurations for energy efficiency saw 26-41% reduction in battery consumption. This can be attributed to the fact that the *powersave* governor scheduled the majority of JS Web API executions to the big cluster at lowest frequency, while only allocating 16.5% to the LITTLE cluster, as shown in Figure 8a.

Additionally, we observed that the optimal configurations for performance outperformed the *performance* governor by 4-27% in terms of API execution duration. Figure 8b shows that the *performance* governor scheduled 82.6% of API executions to the big cores at high frequencies, while the remaining executions were scheduled to the big cores at low frequencies or to the LITTLE cluster. We also found that the optimal configuration yielded 9-46.2% less EDP when compared to the *schedutil* governor. Contrary to our expectations, as depicted in Figure 8c, the *schedutil* governor heavily utilized the big cluster and high frequencies, which we suspect increased the EDP value by consuming more energy.

5.5 Optimal Configs. per API

Our benchmarks indicate that, on average, each JS Web API has 2 different performance optimization configurations, 3.16 energy optimization configurations, and 3.3 EDP optimization configurations. This suggests that identifying the optimal CPU configuration for executing a given API depends not only on the API itself, but also on various execution-specific factors such as the current state of the DOM tree.

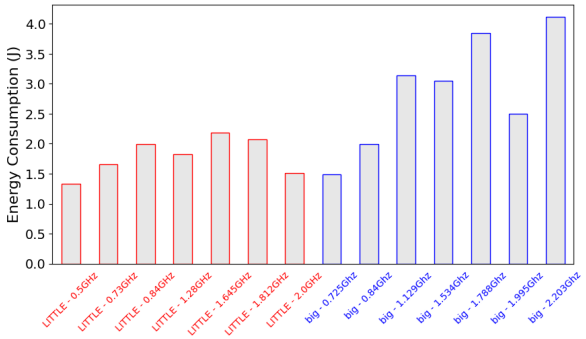


Figure 9: Page-Level Energy Consumption

6 LIMITATION AND FUTURE WORK

Figure 9 depicts the average energy consumption of JS Web API executions per webpage in our benchmarks. These measurements are generally <5% of the total energy consumed during web browsing activities on that page. This implies that optimizing at the JS Web API level may not result in a significant improvement in overall user experience. However, considering that JS Web APIs play a critical role in processing user interactions, the benefits of optimization would increase as the browser handles more user events. Consequently, our future work is to extend the proof of concept presented in this paper into an implementation, possibly as a dynamic scheduler for Chromium's V8 engine. Subsequently, a long-term user study could follow to evaluate the impact of the optimization on both the overall user experience and device battery life.

7 CONCLUSION

This paper presents our findings on the benchmarks of JS Web API executions at various CPU clusters and clock frequencies. The results show that energy consumption could be reduced by running APIs on the median frequencies of the little core, while the big clusters at high clock speed generally result in reduced execution time and EDP. Through a comparison of "optimal" configurations with other configurations and Linux governors, we demonstrate that there is considerable potential for optimization through dynamic frequency scaling and task scheduling across the three key metrics even at the JS Web API level. However, considering that most APIs have short execution times but are frequently called during user interaction, further research is needed