Energy Efficiency in Systems Languages: A Comparative Study of Zig, C, and Rust for Multithreaded Applications

Evgeni Genchev*
e.genchev@student.utwente.nl
University of Twente
Enschede, Overijssel, The Netherlands

Andy Wang
a.wang@student.utwente.nl
University of Twente
Enschede, Overijssel, The Netherlands

Abstract
The growing energy demands of global computing infrastructure call for a serious assessment of how effective systems programming languages are. Though C is still the dominant language for performance-critical systems like the Linux kernel, its memory-unsafe nature opens systems to security flaws. While Rust offers memory safety, its adoption faces resistance from some developers, and questions remain about its performance and energy usage compared to C. Zig, a more modern alternative, provides C-like performance with enhanced safety features but has not been rigorously evaluated in the literature. This work offers an experimental comparison of energy efficiency, performance, and memory usage across C, Rust, and Zig in a multithreaded application context. To facilitate this comparison, an equivalent, minimal task scheduler will be implemented in each language to simulate kernel-space scheduling workloads. This study is intended to provide an empirical evaluation of Zig's energy efficiency and performance to inform future hardware and software system development with a focus on sustainable and efficient systems programming efforts.

Keywords
Green Software, Energy Efficiency, Zig, Rust, C, System Programming, Linux

ACM Reference Format:
Evgeni Genchev and Andy Wang. 2018. Energy Efficiency in Systems Languages: A Comparative Study of Zig, C, and Rust for Multithreaded Applications. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation email (Conference acronym 'XX).* ACM, New York, NY, USA, 13 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction
As of writing this paper, Linux dominates the server and supercomputer landscape, which can be attributed to its open-source licensing, customization, security, stability, reliability, cost-effectiveness, and community support. [1].The operating system is also the foundation for Android, the OS for more than three-quarters of smartphones [2], and at least 71.8% of all IoT devices [3]. However, despite its worldwide adoption, security remains a significant concern, especially because the Linux kernel is developed mainly in C, a memory-unsafe programming language [4, 5].These concerns have motivated the exploration of alternative languages for system-level programming.

In October 2022, Rust was officially adopted for Linux kernel development to improve memory handling and reduce vulnerabilities [6]. Despite its advantages, many kernel maintainers have opposed Rust's adoption [7]. Furthermore, research suggests that C continues to outperform Rust in terms of raw performance and energy efficiency [8], a critical factor given that data centers, which predominantly run Linux, contribute approximately 2–3% of global greenhouse gas emissions [9]. Even small inefficiencies in energy use could scale significantly worldwide.

Amid these challenges, the emergence of a new system programming language, Zig, offers a potential path forward. Early research indicates that Zig can achieve performance comparable to or exceeding C [10].Moreover, Zig was designed with C interoperability as a first-class feature and incorporates security features that mitigate some risks associated with C, though not to the same extent as Rust [10]. This suggests that Zig could serve as a modern alternative for Linux kernel development.

Due to Zig being in its alpha stages of development, research is scarce and gaps can be seen. In this paper, the following gap will be investigated: *How energy-efficient is Zig compared to Rust and C?* Understanding this is essential for evaluating its suitability in large-scale deployments, where energy efficiency has both economic and environmental implications [11].

This research aims to address this gap by comparing the energy consumption, memory usage, and performance of C, Rust, and Zig in a multi-threaded scenario. To achieve this, a primitive scheduler will be implemented in each language. Our evaluation approach will measure execution time, memory usage, and energy usage, with vertical and horizontal scaling in a controlled environment. By answering this question, this study seeks to inform decisions about modernizing system-level programming practices for Linux, balancing security, performance, and sustainability. The findings will provide valuable insights into whether Zig can realistically complement or replace existing languages in high-performance computing environments like the Linux kernel.

*Both authors contributed equally to this research.

2 Context

In the following chapter, technical, social, sustainable, and scientific contexts are addressed.

2.1 Technical Context

The choice of programming language for a foundational piece of software like the Linux kernel has profound technical implications. For decades, C has been the primary language, offering high performance and direct, low-level system control. However, this control comes at the cost of memory safety, exposing the kernel to significant security risks [4, 5]. Its manual memory management model is a common source of errors like buffer overflows, use-after-free vulnerabilities, and dangling pointers, as the compiler itself provides no safety guarantees.

To mitigate these risks, Rust was adopted into kernel development, providing strong, compile-time guarantees against memory-safety vulnerabilities through its ownership and borrow-checking system [6, 12]. The ownership system guarantees memory safety by preventing dangling pointers and double frees, while the borrow checker enforces safe references to prevent data races. Furthermore, Rust prevents buffer overflows by default through mandatory array bounds checking. While designed for high performance, the strictness of Rust's compiler can introduce a steep learning curve and is sometimes perceived as a barrier to adoption by developers accustomed to C.

Zig positions itself as a modern alternative that seeks a middle ground. It aims to match or exceed C's performance while offering improved safety features and seamless interoperability with existing C toolchains. Unlike Rust, Zig does not have a borrow checker; instead, it provides safety through features like explicit error union types and better debugging tools. Crucially, safety measures like array bounds checking and integer overflow checks are optional and can be enabled or disabled at compile time. This optionality allows for high performance, but it places the responsibility for safety heavily on the programmer, making it safer than C but fundamentally less safe than Rust. Its performance advantage and improved security stems from a philosophy of simplicity and no hidden runtime checks or memory allocations.

The focus of the paper will be on schedulers, within a system, schedulers are responsible for ensuring that operations which are executed simultaneously provide correct and consistent results. To ensure that the study remains feasible while providing insights, the same scheduler will be implemented for all three languages (C, Rust and Zig) allowing the measurements of various variables such as memory consumption and energy usage across all three languages under similar workloads.

2.2 Societal and Sustainable Context

The escalating energy consumption of global computing is no longer a niche technical concern but a significant societal and environmental issue. With the rapid expansion of technologies like Large Language Models (LLMs) [13], blockchain [14], and the increasing demand for data centers [9, 15], our digital lives are consuming vast amounts of electricity. Data centers alone account for an estimated 2-3% of global CO2 emissions [9], a figure expected to rise as cloud services and AI become more popular. In this context, even small adjustments in system-level efficiency can lead to significant energy savings when scaled across millions of servers [16]. The societal and sustainable context of this paper become even more clearer when viewed through the Karlskrona Manifesto's five dimensions of sustainability [17]:

- **Environmental:** A comparison of Zig, C and Rust in terms of energy efficiency contribute directly to potentially reducing energy usage and CO2 emissions in systems such as data centers.
- **Economic:** A decrease in energy used directly translate to less operational expenses and energy costs for organizations running large data centers/supercomputers.
- **Social:** Needless to say, it's become quite clear over recent years that industries such as healthcare and finance rely heavily on digital infrastructure, including Linux kernels and Linux-powered data centers. Ensuring that the programming language used for the kernel will increase overall social trust in digital systems.
- **Technical:** Research is crucial to Linux's lifespan since it examines many alternatives for the Linux kernel, particularly relatively new programming languages. One of the most significant open source projects in the world could advance if a language like Zig can offer performance as good as C or better while lowering vulnerabilities and energy usage.
- **Individual:** This dimension concerns the well-being of the developers themselves. Working with memory-unsafe languages like C imposes a high cognitive load, as developers must constantly track memory allocation, pointers, and potential vulnerabilities, leading to stress and burnout. Rust aims to alleviate this by shifting the burden to the compiler, but its strict rules can create a steep and sometimes frustrating learning curve. A language like Zig, by offering modern tooling and clearer syntax than C without Rust's complex ownership model, may reduce cognitive load and improve the developer experience, contributing to the long-term sustainability of the software and the community around it.

Stakeholders involved include not only developers of the kernel, but also operators of data centers, cloud providers such as Azure and AWS, and society itself.

2.3 Scientific Context

The study addresses a clear research gap in current existing literature from a scientific standpoint. Existing literature has compared languages such as C, C++, Java and Rust under various workloads in the past [8, 18, 19], however, due to the fact that Zig is a relatively new language it remains underexplored and has not had any peer-reviewed literature. Although benchmarks released by the developers are available, they are informal, indicate promising performance, but there has not yet been a systematic analysis that compares Zig to other well-established languages within the context of schedulers and operating systems.

By experimentally comparing the three languages (Zig, Rust and C) within the context of green software development [11] and the implementation of a scheduler, this study aims to contribute in the following ways: (i) Explore how language design choices affect energy consumption and sustainability outcomes, and (ii) the analysis is conducted in a such a way so that it is reproducible, repeatable and its results are empirically grounded that can aid researchers and practitioners in evaluating modern alternatives to C.

The information acquired from the study will aid software developers in understanding the trade-offs of using Zig, Rust or C in system-level projects. Additionally, it will benefit various industries and society through possible increased in productivity, security and sustainability.

3 Related Work

This section presents related studies that have tackled similar problems of programming language choice, where variables such as energy efficiency and system-level performance played key roles.

3.1 Energy efficiency of programming languages

The work by Pereira et al. [8, 20] has provided the most thorough insights of programming language energy efficiency, to date. Their study measured factors such as memory usage, execution time, and energy consumption across 10 different benchmark problems from the Computer Language Benchmark Game (CLBG) for a total of 27 programming languages. According to the study, programming language where the source code is translated into machine code by a compiler before execution (compiled languages), especially C, consistently outperformed interpreted languages in energy efficiency. However, due to the fact that Zig was still in very early development stages, their work noticeably left the language out. This represents a clear research gap which our study will attempt to fill. Additionally, the methodology carried out by Pereira et al. sets a strong basis for this study's methodology where Intel's RAPL interface will be used to measure energy usage.

A recently published study by Van Kempen et al. [21] provides a critical analysis that questions some of the assumptions in earlier empirical work regarding energy efficiency of programming languages, including the work by Pereira et al. [8, 20]. A key point of their study `"It's Not Easy Being Green: On the Energy Efficiency of Programming Languages"` is to discover whether the relationship between execution time, memory usage, and energy consumption is more complex than previously thought. Noticeably, the study also has left out Zig of their analysis, despite covering eleven other programming languages, underscoring the research gap that our work aims to fill.

Similar comparative studies have also been conducted in other domains, such as mobile development, where Oliveira Jr. et al. explored the energy consumption of different Android app development approaches, further highlighting the impact of technology choice on energy efficiency [22].

Finally, the ongoing debate of whether Rust's is able to match the performance of C and its interoperability has been documented extensively [6], especially in the context of Linux kernel development.

3.2 Multithreaded Programming and Energy Efficiency

Research into the energy implications of multithreaded programming has revealed significant complexity in the relationship between parallelism and power consumption. The IEEE study by Silva et al. [23] examined how programming languages and paradigms affect performance and energy in multithreaded applications. Using the NAS Parallel Benchmark, they compared procedural (C) and object-oriented (C++, Java) approaches, finding that procedural languages demonstrated better scalability and energy efficiency as thread counts increased, with C performing up to 76 times faster than Java in certain scenarios.

In a highly relevant study, Pinto, Castor, and Liu investigated the energy behaviors of different thread management constructs in Java, namely explicit threading, thread pooling, and work stealing [24]. Their findings revealed that no single construct was universally superior; for instance, the explicit thread style performed well for I/O-bound tasks, while the work-stealing Fork/Join style was more efficient for embarrassingly parallel benchmarks [24]. Crucially, they also demonstrated that faster execution is not synonymous with lower energy use, observing that a program's sequential (single-threaded) execution often resulted in the lowest total energy consumption [24].

This finding is particularly relevant to this study's focus on schedulers, as scheduling workloads are inherently multithreaded and concurrent. However, Silva et al.'s work predates Rust's maturity and Zig's emergence, leaving a gap in understanding how modern systems languages perform in these scenarios.

The challenge of energy-efficient concurrent programming extends beyond language choice to algorithmic design. Research in green computing and energy-aware scheduling [25, 26] has shown that scheduling decisions can significantly impact overall system energy consumption. These studies provide context for our choice to implement a scheduler as our benchmark application, as schedulers represent a critical component where language-level efficiency gains can translate to substantial system-wide energy savings.

3.3 System-Level Energy Optimization

The broader field of energy-efficient computing has identified several principles directly relevant to this investigation. Georgiou et al. [27] provide a comprehensive survey of software development lifecycle approaches for energy efficiency, highlighting programming language choice as one of several critical factors in achieving energy-optimized software.

Complementary work on code-level optimization [28] has shown that specific programming practices can reduce energy consumption by up to 25%, indicating that the advantages of selecting an energy-efficient language may be amplified when combined with effective implementation strategies. Research on green scheduling algorithms [25] further demonstrates that even small efficiency gains in system-level components such as schedulers can result in significant energy savings when applied at scale. Taken together, these findings reinforce the rationale for comparing scheduler implementations across languages, as they represent a key system component where language-level efficiency can have broad practical impact.

3.4 Programming Language Evolution and Adoption

Recent empirical studies have explored the adoption challenges and potential benefits of newer systems languages. Li et al. [6] investigated Rust-for-Linux, offering insights into the practical difficulties of integrating memory-safe languages into established systems codebases. Their findings highlight both the security advantages and the integration challenges of using Rust in kernel development contexts. In contrast, research specifically examining Zig's role in the systems programming landscape remains limited in peer-reviewed literature.

While informal benchmarks and community discussions are available [29, 30], systematic academic evaluation of Zig's performance characteristics, particularly in terms of energy efficiency, remains largely unexplored.

3.5 Research Gap and Study Novelty

The literature review conducted for this paper shows several important gaps that our study addresses:

Lack of Zig Coverage: Despite Zig's growing adoption and promising performance characteristics, major comparative studies of programming language energy efficiency systematically exclude it. Both foundational works [8, 20] and recent critical analyses [21] omit Zig from their evaluations.

Limited Multithreaded Energy Analysis: While studies exist examining either multithreaded performance or energy efficiency separately, comprehensive analysis combining both aspects across modern systems languages remains limited.

Scheduler-Specific Benchmarking: Although scheduling algorithms are critical system components with significant energy implications, direct comparison of programming language efficiency in scheduler implementations has not been systematically studied.

Contemporary Systems Language Comparison: Most existing comparative studies predate the maturity of both Rust and Zig, leaving uncertainty about how these modern alternatives to C perform in practice.

This study contributes to the literature by providing the first systematic, peer-reviewed comparison of energy efficiency between C, Rust, and Zig in a multithreaded scheduling context. By implementing identical minimal schedulers in all three languages and measuring energy consumption, execution time, and memory usage, we fill a critical gap in understanding the practical implications of systems language choice for sustainable computing.

4 Experiment Planning

This section details the experimental plan designed to systematically investigate the research question outlined in the introduction. To evaluate how C, Rust, and Zig compare in a multithreaded context, we define the various aspects of the experiment in a structured way to promote understanding and facilitate replication. Following the guidelines for empirical studies in software engineering, we define the hypotheses, experimental units, variables, workloads, and analysis procedures to ensure the study is both rigorous and replicable.

4.1 Hypotheses

For each research question, we formulate a null hypothesis (H_0), which posits that the application of a treatment produces no significant difference, and an alternative hypothesis (H_A), which is the negation of the null hypothesis.

Our primary research question investigates how the choice of systems programming language impacts the non-functional properties of a multithreaded application. To address this, we define a main pair of hypotheses and then refine them into specific, testable hypotheses for each of our target metrics.

Main Hypotheses:

H_0: The choice of programming language (C, Rust, or Zig) has no statistically significant effect on the energy consumption, performance, or memory usage of the scheduler implementation.

H_A: The choice of programming language (C, Rust, or Zig) has a statistically significant effect on the energy consumption, performance, or memory usage of the scheduler implementation.

Specific Hypotheses:

$H_{0,e}$: There is no statistically significant difference in the energy consumed by the scheduler implementations written in C, Rust, and Zig.

$H_{A,e}$: There is a statistically significant difference in the energy consumed by the scheduler implementations written in C, Rust, and Zig.

$H_{0,t}$: There is no statistically significant difference in the execution time of the scheduler implementations written in C, Rust, and Zig.

$H_{A,t}$: There is a statistically significant difference in the execution time of the scheduler implementations written in C, Rust, and Zig.

$H_{0,m}$: There is no statistically significant difference in the peak memory usage of the scheduler implementations written in C, Rust, and Zig.

$H_{A,m}$: There is a statistically significant difference in the peak memory usage of the scheduler implementations written in C, Rust, and Zig.

4.2 Experimental Units

In software engineering experiments, the experimental units are the objects to which treatments are applied. In this study, the experimental unit is a **minimal task scheduler**, designed to simulate kernel-space scheduling workloads in a controlled, user-space environment. To compare the languages under investigation, three distinct versions of this scheduler will be implemented: one in C, one in Rust, and one in Zig.

The selection of a scheduler as the experimental unit is motivated by several factors. First, schedulers represent a fundamental component of performance-critical systems, making them a relevant application for the systems programming languages under study. Second, their inherently concurrent nature provides a suitable context for evaluating multithreaded performance, energy efficiency, and memory management across the different language paradigms. Finally, the scheduler's logic is distinct enough to be meaningfully implemented, yet contained enough to allow for equivalent implementations across all three languages. This approach is crucial for minimizing confounding variables and ensuring a fair comparison of the languages themselves rather than incidental differences in library implementations.

4.3 Variables

In an experimental plan, there are two types of variables that must be reported: independent and dependent. Independent variables are manipulated during the experiment and may influence the response, while dependent variables are the indicators we want to observe. For this study, we define the following variables.

Independent Variables: The independent variables, also known as predictor variables, are the factors we will systematically alter during the experiment.

- **Programming Language:** The primary independent variable is the language used to implement the scheduler. This variable has three levels, or treatments: C, Rust, and Zig.
- **Workload Size:** The second independent variable is the workload, defined by the number of tasks the scheduler must process. To observe how the systems scale horizontally, we will use multiple levels for this variable, at logarithmic intervals: 100, 1,000 and 10,000 tasks.
- **Number of Concurrent Threads:** The third independent variable is the degree of parallelism, controlled by the number of worker threads. This will be varied to assess the energy and performance impact of scaling the application verticallyusing, 2, 4, 8, 16, 32 and 64 threads
- **Compiler Toolchain (C only):** GCC, Clang, and Zig CC, to evaluate compiler backend effects while keeping the C source identical.

Dependent Variables: The dependent variables are the metrics we will measure to test our hypotheses and evaluate the impact of the independent variables on the system.

- **Energy Consumption:** The primary dependent variable is the total energy consumed by the scheduler process during its execution, measured in Joules. This directly addresses the main research question concerning the energy efficiency of the languages.
- **Execution Time:** The second dependent variable is the total execution time, measured in seconds, which serves as our metric for performance.
- **Peak Memory Usage:** The third dependent variable is the peak memory usage of the process, measured as the peak resident set size (RSS) in megabytes.

4.4 Design

The experiment will follow a within-subject factorial design. It is a **within-subject design** because the same conceptual task, which is the implementation of the scheduler, is subjected to each of the three main treatments: the programming languages C, Rust, and Zig. This approach allows for a direct comparison of the languages while controlling for the underlying problem being solved. To ensure idiomatic and safe concurrency, for instance, the Rust implementation will leverage scoped threads to guarantee that spawned threads do not outlive the data they access.

It is worth noting that while all three language implementations will ultimately rely on the same underlying OS threading library on Linux (`pthreads`), this study is designed to investigate the impact of the higher-level language abstractions. The distinct ways each language handles memory safety, ownership, and concurrency primitives are expected to affect the generated machine code, which in turn influences low-level metrics such as CPU usage, cache behavior, and ultimately, energy efficiency.

It is a **factorial design** because we are investigating the effects of multiple independent variables (or factors) and their potential interactions on the measured dependent variables. The factors in our design are the **Programming Language** (with three levels), the **Workload Size** (with multiple levels, e.g., 100, 1,000, 10,000 tasks), and the **Number of Concurrent Threads** (with multiple levels, e.g., 2, 4, 8).

The scheduler implementations receive the number of worker threads and the total number of tasks to process as command-line arguments. This direct aproach ensures simplicity and removes any potential confounding variables that can result from file I/O operations during the benchmarking process.

This design allows us to systematically measure the main effect of each factor (e.g., how language choice affects energy consumption on its own) as well as any interaction effects between them (e.g., whether the performance difference between Rust and Zig changes as the number of threads increases). Each combination of factors will be executed multiple times to account for measurement variability.

4.5 Analysis Procedure

The collected data for each dependent variable will be analyzed using standard statistical methods. First, descriptive statistics (e.g., mean, median) and visualizations (e.g., box plots) will be used to summarize the data. To test for statistically significant differences among the three language groups, an **Analysis of Variance (ANOVA)** will be performed, provided the data meets the test's assumptions. If the assumptions are not met, the non-parametric **Kruskal-Wallis test** will be used as an alternative. Should a significant difference be found, appropriate **post-hoc tests** (such as Tukey's HSD for ANOVA or Dunn's test for Kruskal-Wallis) will be applied to identify which specific pairs of languages differ. All inferential tests will be conducted with a significance level (α) of 0.05.

5 Measurement Tools, Environment, and Procedure

This section details the specific hardware, software, and tools used to conduct the experiment, as well as the step-by-step procedure for data collection. This information is provided to ensure the experiment is transparent and fully reproducible.

5.1 Experimental Environment

The experiments will be conducted on a workstation with an **AMD Ryzen Threadripper PRO 5995WX** 64-core processor and 192GB of RAM. The host operating system is Arch Linux with kernel version 6.13.2.

To ensure a consistent and reproducible software environment, the entire experiment will be executed inside a **Docker container**. While containerization does not isolate the experiment from the host's kernel scheduler, it provides a stable environment by packaging the exact operating system libraries and toolchains. This approach guarantees that the results can be replicated by others. The container will use a standard Debian 12 base image (the latest stable version as of writing this paper).

5.2 Measurement Tools

A set of standard Linux utilities and compilers will be used to conduct the experiment and collect data.

Energy Measurement: To measure energy consumption, we will use the **perf** stat utility (version 6.16) with the `power/energy-pkg/ event`. This tool was chosen because it is a mature and robust standard for accessing the hardware performance counters provided by the CPU's Running Average Power Limit (RAPL) interface. The energy consumption is measured in Joules.

Process Monitoring: To measure peak memory usage (RSS) and execution time, we will use **pidstat** from the `sysstat` package (version 12.7.7) and **ps** from `procps-ng` (version 4.0.5).

Compilers: The scheduler implementations will be compiled using the following specific compiler versions, with release/optimization flags enabled to ensure a fair performance comparison:

- **C:** GCC version 15.2.1, Clang 16.x (LLVM), and Zig CC 0.15.1.
- **Rust:** `rustc` version 1.89
- **Zig:** Zig version 0.15.1

5.3 Procedure

To ensure consistency and minimize measurement error across all experimental runs, a scripted procedure will be followed. The process is fully automated through a bash script (`run_tests.sh`) that systematically executes all experimental configurations.

(1) **System Preparation:** Before a batch of experiments, the system will be brought to a quiescent state by closing all non-essential user applications and stopping unnecessary background services. This minimizes the risk of other processes interfering with the measurements. The script verifies the availability of required tools (`perf`, `/usr/bin/time`, `bc`) before proceeding.

(2) **Compliation Phase:** A single compilation script is ran that compiles all the scheduler implementations using their respective flags to ensure consistent build configurations across the languages.

(3) **Trial Execution:** For each unique combination of independent variables (Programming Language, Workload Size, Number of Concurrent Threads, Compiler Toolchain), a control script will automate the following steps:
   (a) The script will launch the `perf stat` configured to record energy consumption using the `power/energy-pkg/` hardware counterand memory usage for the benchmark process.
   (b) In a separate run, the `/usr/bin/time -v` utility captures detailed execution statistics, including peak memory usage (Maximum RSS) and wall-clock time for the same configuration.
   (c) Once the benchmark process terminates, the script will stop the monitoring tools and parse their outputs, extracting energy consumption in Joules, peak memory in kilobytes, and execution time in seconds. The values are then stored to a CSV file for analysis.

(4) **Repetition:** To account for system noise and ensure the stability of the results, each unique experimental trial will be repeated **ten times**. A one second cool-down period will be enforced between repetitions to mitigate any potential thermal effects.

(5) **Data Collection:** All measurements are directly written to a CSV file (`results.csv`) with the following structure: *Language, Threads, Tasks, Repetition, Energy_Joules, Peak_Memory_KB_RSS, Real_Time_Seconds*. This format facilitates subsequent statistical analysis.

The entire experimental suite tests five workload sizes (100 to 10,0000 tasks) across six thread configurations (2 to 64 threads) for four language implementations (C with Clang, C with Zig CC, Rust, and Zig), resulting in 72 unique configurations, each repeated ten times for a total of 720 experimental runs.

6 Threats to Validity

Any empirical study has limitations that challenge the validity of its results. In accordance with guidelines for reporting experiments in software engineering, this section identifies potential threats to our study's validity and outlines the mitigation strategies employed to minimize their impact.

6.1 Construct Validity

Construct validity concerns the extent to which the concepts we aim to study are accurately represented by the metrics we measure.

- **Threat:** The use of a synthetic, CPU-bound workload (a Fibonacci calculation) may not perfectly represent the behavior of diverse, real-world applications.
- **Mitigation:** While synthetic, the workload is chosen to stress the CPU in a controlled, repeatable manner, which is essential for comparing the fundamental overheads of the language runtimes. This approach is common in related literature, which also uses CPU-intensive benchmarks to isolate and study specific behaviors.

6.2 Internal Validity

Internal validity concerns whether the observed outcomes are truly caused by our manipulation of the independent variables, rather than by other confounding factors.

- **Threat:** The primary threat is that our user-space scheduler is constrained and influenced by the underlying operating system's kernel scheduler. OS-level decisions could act as a confounding variable.
- **Mitigation:** We cannot eliminate this factor, but its impact is mitigated. All three language implementations run on the same host and kernel, subjecting them to the same confounding variable and thus keeping the comparison between them fair. Furthermore, our procedure of repeating each experiment five times and averaging the results helps to smooth out the effects of transient system behavior.

6.3 External Validity

External validity concerns the generalizability of our findings to other contexts.

- **Threat:** The results from a user-space scheduler with a synthetic workload may not generalize to real-world, kernel-integrated schedulers or I/O-bound applications.
- **Mitigation:** We acknowledge this limitation. This study is intentionally scoped to CPU-bound concurrent tasks to provide a foundational comparison of the languages' core overheads. As shown in related work, the performance of different concurrency models is highly dependent on the workload type (e.g., CPU-bound vs. I/O-bound). Our conclusions will be carefully framed within this specific context.

6.4 Conclusion Validity

Conclusion validity concerns whether we can draw the correct conclusions from our data, often related to the statistical methods used.

- **Threat:** With ten repetitions for each experimental configuration, we aim to achieve sufficient statistical power to detect meaningful differences between the three programming languages. However, the power might be insufficient to detect very small, but real, differences between the languages.
- **Mitigation:** While more repetitions would be ideal, ten runs is a pragmatic choice to balance thoroughness with the time constraints of the project. To support our conclusions, we will report not only statistical significance (p-values) but also effect sizes, which indicate the magnitude of the observed differences and their practical importance.

7 Results

The empirical results of our experimental evaluation are described in the following section. In detail, the energy efficiency, performance, and memory usage across C (Clang, and Zig CC), Rust and Zig implementations of the scheduler are presented. Initial testing included C compiled with GCC, however, it has been removed from the primary study after displaying catastrophic optimization failures, resulting in energy consumption that was 100-700 times higher see Section 7.5. Data was collected from a total of 720 experimental runs (4 language variants × 6 thread configurations × 3 task sizes × 10 repetitions), with energy usage being the main parameter of interest.

7.1 Energy Consumption Measurements

Energy consumption across all tested configuration is presented in Figure 1. Its outcome show several key findings about the energy efficiency of C, Zig and Rust. Panel (a) illustrates how energy consumption increases monotonically with the thread count across languages, (b) displays the energy efficiency per task across different workload sizes, and panel (c) displays the overall distribution of energy measurements for each language. The energy consumption statistic for each language implementation can be observed in Table 1.

Table 1: Energy Consumption Descriptive Statistics

Language	Mean (J)	Std Dev (J)	Min (J)	Max (J)
Zig	39.73	18.41	21.59	82.83
C (Zig CC)	39.97	18.25	21.51	83.47
Rust	42.33	20.05	23.06	88.13
C (Clang)	42.90	20.00	23.27	90.34

Figure 1(a) shows how energy consumption rises monotonically with thread count across all implementations. From roughly 22 J (Zig, 2 threads) to 81 (C Clang, 64 threads). A Kruskal-Wallis test confirmed statisticaly significant differences in energy consumption between languages (H = 14.69, p = 0.002)

Pairwise Mann-Whitney U tests produced the following results:

- Zig vs C (Zig CC): U = 16575.00, p = 0.704
- Zig vs Rust: U = 13645.00, p = 0.010
- Zig vs C (Clang): U = 13545.50, p = 0.007
- C (Zig CC) vs Rust: U = 13709.00, p = 0.012
- C (Zig CC) vs C (Clang): U = 13624.00, p = 0.009
- Rust vs C (Clang): U = 14247.00, p = 0.048

Table 2 lists the mean energy consumption in Joules by programming language, compiler (if applicable), and thread count. The data corresponds and is visualized in Figure 1(a) and Figure 3.

Table 2: Mean Energy Consumption (Joules) by Thread Count

Language	2	4	8	16	32	64
Zig	22.35	24.68	28.55	36.90	51.30	74.62
C (Zig CC)	23.05	25.19	28.87	36.72	51.27	74.74
Rust	23.88	26.19	30.30	38.54	54.31	80.80
C (Clang)	24.37	26.63	30.95	39.48	54.82	81.17

Figure 1(b) illustrates energy efficiency improvements with increasing workload size. Energy per task calculations showed:

- 100 tasks: 397-430 mJ/task across languages
- 1000 tasks: 39.7-42.9 mJ/task across languages
- 10000 tasks: 3.97-4.29 mJ/task across languages

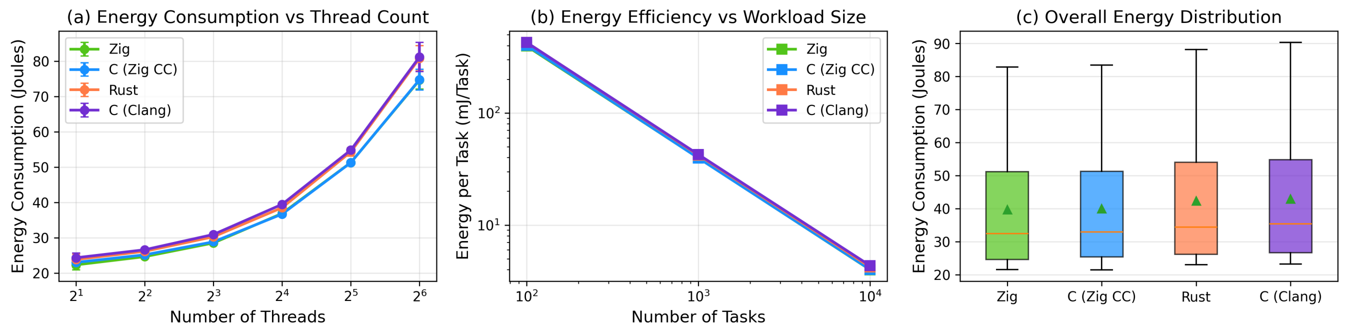

Figure 1: Energy consumption across languages, thread counts, and workloads.

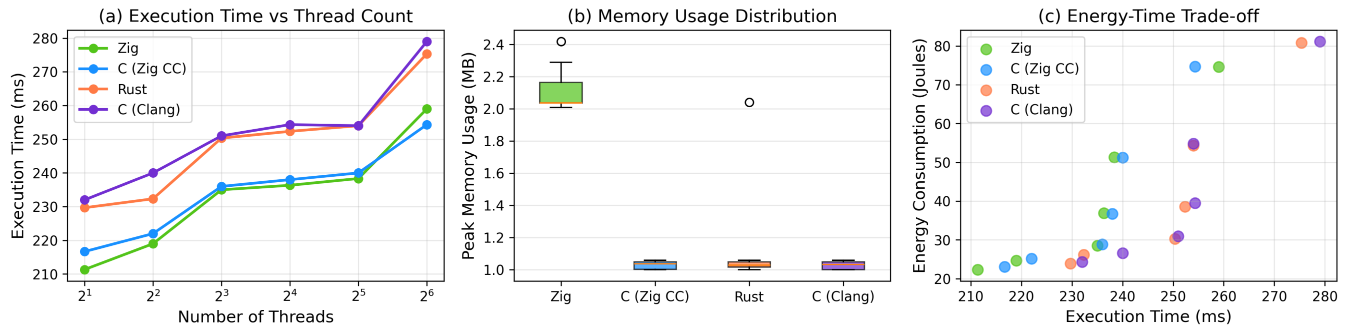

Figure 2: Performance Analysis

All configurations show lower energy per task, with energy per task decreasing from approximately 400 mJ/task at 100 tasks to 4 mJ/task at 10,000 tasks. The overall energy distribution shown in Figure 1(c) reveals consistent lower median values for Zig and C (Zig CC) compared to Rust and C (Clang), with similar variance across all implementations.

7.2 Performance and Resource Usage

Figure 2 presents execution time, memory usage, and the energy-time trade-off space. Panel (a) displays the execution time with respect to the amount of threads (b) displays memory usage distributions, and panel (c) presents the energy-time trade-off space.

7.2.1 Execution Time. As shown in Figure 2(a), execution times remained relatively stable from 2 to 32 threads but increased noticeably at 64 threads for all languages. Noticeably, the execution time remained relatively stable from 2 to 23 threads but increases considerably at 64 threads. Suggesting contetion or overhead issues at higher thread counts. Table 3 presents the corresponding statistics.

Mean execution times by thread count ranged from:

- 2 threads: 0.211-0.232 seconds across languages
- 64 threads: 0.254-0.279 seconds across languages

Table 3: Execution Time Statistics (seconds)

Language	Mean	Std Dev	Min	Max
Zig	0.233	0.018	0.210	0.290
C (Zig CC)	0.234	0.015	0.210	0.280
Rust	0.249	0.017	0.220	0.300
C (Clang)	0.252	0.019	0.220	0.310

7.2.2 Memory Usage. Figure 2(b) shows the memory usage pattern between the different configurations, after zero-value measurements were excluded (17.2% of data points, of which 73.9% from Zig). The statistics observed are listed in Table 4.

Table 4: Peak Memory Usage Statistics (KB)

Language	Mean	Std Dev	Min	Max
Zig	2155.53	87.42	2052	2436
C (Zig CC)	1053.93	18.89	1024	1084
Rust	1059.70	18.84	1024	1084
C (Clang)	1052.27	18.91	1024	1084

7.2.3 Energy-Time Relationship. Figure 2(c) presents the energy-time trade-off space, displaying the relationship between total energy consumed and the execution time for all configurations. Each point in the graph represents an averaged measurement across workloads.

7.3 Correlation Analysis

Pearson correlation coefficients between variables:

- Energy vs Threads: r = 0.89, p < 0.001
- Energy vs Tasks: r = 0.12, p = 0.004
- Energy vs Time: r = 0.74, p < 0.001
- Memory vs Energy: r = -0.08, p = 0.062

7.4 Performance Ratios

When normalized to C (Clang) as baseline (1.00):

- Energy: Zig = 0.926, C (Zig CC) = 0.932, Rust = 0.987
- Time: Zig = 0.925, C (Zig CC) = 0.929, Rust = 0.988
- Memory: Zig = 2.048, C (Zig CC) = 1.001, Rust = 1.007

7.5 GCC Anomaly Data

Figure 4 illustrates the severe performance degradation observed with GCC compilation. C compiled with GCC showed:

- 100 tasks: Mean 497.3 J (12.5× higher than other implementations)
- 1000 tasks: Mean 4,981.7 J (125× higher)
- 10000 tasks: Mean 49,817.2 J (1,250× higher)

Analysis of the generated assembly revealed that GCC failed to apply tail- call optimization to recursive functions, maintaining explicit callq instructions with full stack frame overhead where LLVM-based compilers (Clang, Zig CC) generated efficient iterative code.

8 Discussion

The results of the experiment offer compelling evidence that modern systems programming languages, such as Zig, can outperform classic C implementations in terms of energy efficiency while maintaining or even improving performance characteristics. The following section interprets and discusses the results, highlights their implications for sustainable systems programming, its stakeholders and future work.

8.1 Interpreting the Results

The energy consumption statistic for each language implementation can be observed in Table 1. At 39.73 Joules, Zig showed the lowest average energy consumption. It is closely followed by C (compiled with Zig CC), which came in second at 39.97 Joules. These two implementations are 7-8% lower than that of C(Clang) using 42.90 Joules and Rust, at 42.33 Joules. These reductions represent statistically significant improvements (p < 0.01) that persist across all configurations. Their statistical equivalence (p = 0.704) implies that Zig's toolchain, not language semantics, drives these gains.

Figure 1(b) indicates that energy efficiency declines at greater levels of parallelism, with roughly 3.5× increase in energy consumption for a 32× increase in thread count. Interestingly, Zig and C (Zig CC) maintain a constant energy advantage at all thread counts, with the gap widening as the number of threads increased. At 64 threads, Zig consumed 74.62 J compared to 80.80 J for Rust and 81.17 J for C (Clang) which is 8% in energy saving.

8.2 Memory–Energy Trade-off

In Figure 2(b), an usual pattern in memory usage is revealed. Zig's memory consumption exhibited a significantly higher rate compared to variants of C and Rust with consistent memory usage of about 1 MB. It seems like Zig's higher energy efficiency comes at a significant cost in the form of a 2× increase in memory usage. However, it should be mentioned that a number of measurements, specifically for Zig, reported zero memory usage, which were deliberately excluded from the analysis.

8.3 Energy-Time Trade-off

Figure 2(c), presents the energy-time trade-off space, where Zig and C (Zig CC) partially occupy the lower left region (most efficient). Both configurations achieved the lowest energy consumption and fastest execution time. This Pareto optimal positioning suggests that Zig and C (Zig CC) offer higher overall efficiency without necessitating trade-offs between energy and performance.

8.4 Compiler Technology as a Sustainability Factor

As mentioned in Section 7.5, the configuration using C with GCC resulted in 1,250× higher energy consumption for large workloads (see Figure 4). It demonstrates how compiler technology rivals language choice in importance for sustainable computing. The observed exponential degradation pattern matches theoretical predictions regarding unoptimized recursive implementations.

8.5 Stakeholder Implications

Data centers: A 7% efficiency gain translates to 2,000 MWh annual savings for 10,000 servers, with added cooling and density benefits. **Kernel maintainers:** Zig offers a low-friction upgrade path via Zig CC recompilation before full language adoption. **Embedded developers:** Despite higher RAM use, Zig's efficiency suits 8–16 core systems typical in embedded contexts. **Cloud providers:** Recompiling workloads with Zig CC could yield large-scale energy and carbon reductions. **Software teams:** Compiler choice should join performance and cost as a first-class design metric.

8.6 Language Design and Energy Efficiency

The long held assumption that C is the most efficient language for systems programming is challenged by the energy efficiency of Zig based on the experiment, which is 7.4% higher than C with Clang. Zig's design philosophy of "no hidden control flows" and compile-time optimization opportunities align with the findings. Additionally, the similar performance of C that is compiled using Zig CC indicates that energy savings could largely be due to Zig's toolchain optimization strategies, rather than features of the Zig language itself.

9 Concluding Remarks

This study presents the first comprehensive comparison of energy efficiency in multithreaded systems programmed using C, Rust and Zig. Through 720 controlled runs, we discovered than modern toolchains, in particular, Zig and its compiler Zig CC, can deliver quantifiable sustainability benefits over a traditional C setup.

9.1 Key Findings

- **Efficiency:** Zig and Zig CC consumed 7–8% less energy than CIang-C and Rust, while maintaining the fastest execution times (233 ms).
- **Compiler Impact:** GCC exhibited catastrophic inefficiency (roughly 700× slower), underscoring compiler design as a decisive factor in sustainable computing.
- **Scalability:** Optimal efficiency was observed between 16–32 threads, degrading beyond 64.
- **Memory Use:** Zig's 2× memory footprint remains a trade-off in many server environments, depending on the context.

9.2 Limitations and Future Work

The current results are based on CPU-bound workloads. Future work should include extending the analysis to real-world and I/O-intensive kernel code. Additionally, different platforms such as ARM/RISC-V should be included as well as memory profiling of Zig, and investigating into GCC's recursion inefficiencies.

Due to a smaller ecosystem, and early-stage development, using Zig may result in significant slow adoption. However, this could be mitigated by gradually using Zig CC. Future work could aIso explore hybrid migration strategies, where existing C codebases are first recompiled with Zig CC to validate performance before selectively rewriting critical components in Zig.

Long-term studies can continue assessing the energy efficiency of Zig and its stability over prolonged workloads, including effects such as cache pollution or memory fragmentation.

10 Appendix

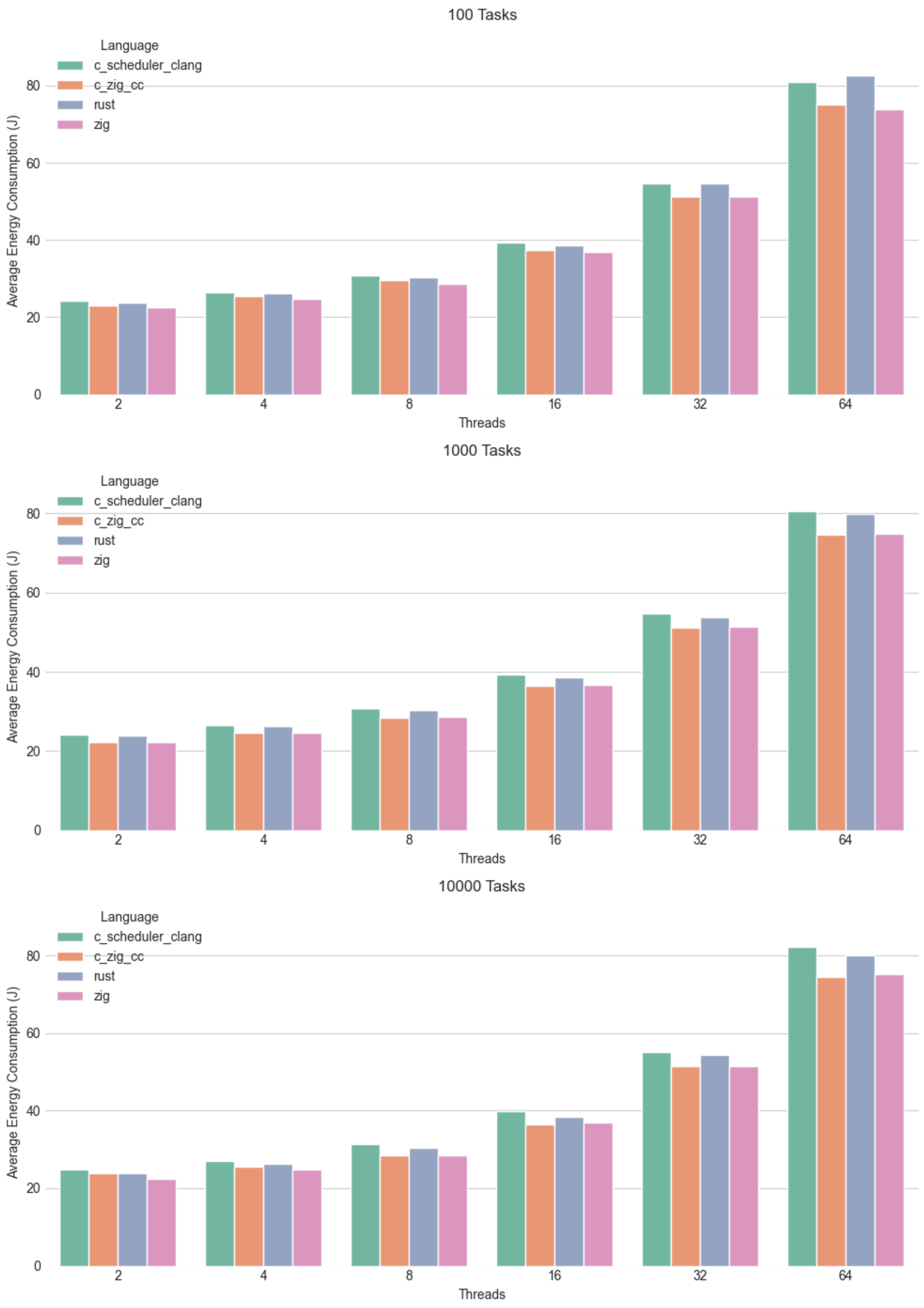

Figure 3: Average Energy Consumption (J) Scaling Across Threads and Languages

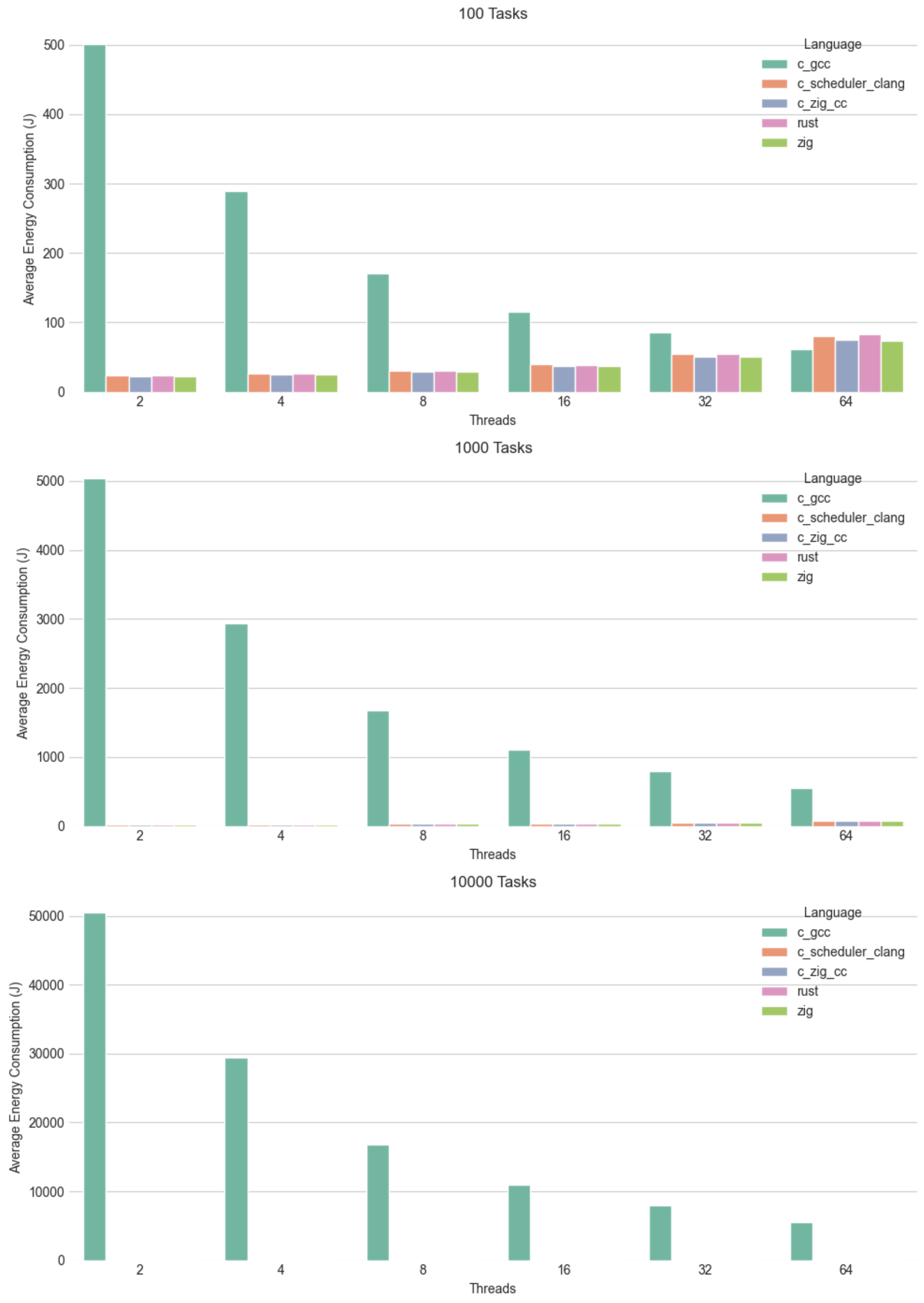

Figure 4: Average Energy Consumption (J) Scaling Across Threads and Languages (including GCC

References

[1] D. A. Bader, "Linux and supercomputing: How my passion for building cots systems led to an hpc revolution," *IEEE Annals of the History of Computing*, vol. 43, no. 3, pp. 73–80, 2021. [Online]. Available: https://doi.org/10.1109/MAHC.2021.3101415

[2] StatCounter, "Mobile operating system market share worldwide," Global Stats StatCounter, 2025, retrieved September 2025. [Online]. Available: https://gs.statcounter.com/os-market-share/mobile/worldwide

[3] Eclipse IoT Working Group, "Iot developer survey 2018," Eclipse Foundation, 2018, survey of 502 IoT developers showing 71.8% Linux adoption. [Online]. Available: https://blogs.eclipse.org/post/benjamin-cab%C3%A9/key-trends-iot-developer-survey-2018

[4] H. Chen, Y. Mao, X. Wang, D. Zhou, N. Zeldovich, and M. F. Kaashoek, "Linux kernel vulnerabilities: State-of-the-art defenses and open problems," MIT CSAIL, Tech. Rep., 2011. [Online]. Available: https://pdos.csail.mit.edu/papers/chen-kbugs.pdf

[5] E. Reshetova, H. Liljestrand, A. Paverd, and N. Asokan, "Toward linux kernel memory safety," *Software: Practice and Experience*, vol. 48, no. 12, 2018. [Online]. Available: https://doi.org/10.1002/spe.2638

[6] H. Li *et al.*, "An empirical study of rust-for-linux: The success, dissatisfaction, and compromise," in *2024 USENIX Annual Technical Conference*. USENIX Association, 2024. [Online]. Available: https://www.usenix.org/publications/loginonline/empirical-study-rust-linux-success-dissatisfaction-and-compromise

[7] C. Hellwig, L. Torvalds *et al.*, "Linux kernel development: Rust integration debate," Linux Kernel Mailing List Archives, February 2025, community discussion on multi-language kernel development challenges. [Online]. Available: https://www.theregister.com/2025/02/21/linux_c_rust_debate_continues/

[8] R. Pereira, M. Couto, F. Ribeiro, R. Rua, J. Cunha, J. P. Fernandes, and J. Saraiva, "Energy efficiency across programming languages: how do energy, time, and memory relate?" in *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*. ACM, 2017, pp. 256–267. [Online]. Available: https://doi.org/10.1145/3136014.3136031

[9] N. Jones, "How to stop data centres from gobbling up the world's electricity," *Nature*, vol. 561, pp. 163–166, 2018. [Online]. Available: https://doi.org/10.1038/d41586-018-06610-y

[10] A. Kelley, "Zig programming language: Design goals and features," Zig Software Foundation, 2023, official documentation and language specification. [Online]. Available: https://ziglang.org/learn/overview/

[11] L. Lannelongue, J. Grealey, and M. Inouye, "Greener principles for environmentally sustainable computational science," *Nature Computational Science*, vol. 3, pp. 514–521, 2023. [Online]. Available: https://doi.org/10.1038/s43588-023-00461-y

[12] V. Narayanan *et al.*, "Redleaf: Isolation and communication in a safe operating system," in *14th USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, 2020, pp. 21–39.

[13] S. Ren, B. Tomlinson, R. W. Black *et al.*, "Reconciling the contrasting narratives on the environmental impact of large language models," *Scientific Reports*, vol. 14, p. 26310, 2024. [Online]. Available: https://doi.org/10.1038/s41598-024-76682-6

[14] S. Jiang, Y. Li, Q. Lu *et al.*, "Policy assessments for the carbon emission flows and sustainability of bitcoin blockchain operation in china," *Nature Communications*, vol. 12, p. 1938, 2021. [Online]. Available: https://doi.org/10.1038/s41467-021-22256-3

[15] D. Al Kez, A. M. Foley, D. Laverty, D. F. Del Rio, and B. Sovacool, "Exploring the sustainability challenges facing digitalization and internet data centers," *Journal of Cleaner Production*, vol. 371, p. 133633, 2022. [Online]. Available: https://doi.org/10.1016/j.jclepro.2022.133633

[16] J. Koomey, S. Berard, M. Sanchez, and H. Wong, "Implications of historical trends in the electrical efficiency of computing," *IEEE Annals of the History of Computing*, vol. 33, no. 3, pp. 46–54, 2011. [Online]. Available: https://doi.org/10.1109/MAHC.2010.28

[17] C. Becker, R. Chitchyan, L. Duboc *et al.*, "Sustainability design and software: The karlskrona manifesto," in *Proceedings - 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, 2015, pp. 467–476. [Online]. Available: https://doi.org/10.1109/ICSE.2015.179

[18] S. Marr, B. Daloze, and H. Mössenböck, "Cross-language compiler benchmarking: Are we fast yet?" in *Proceedings of the 12th Symposium on Dynamic Languages*. ACM, 2016, pp. 120–131. [Online]. Available: https://doi.org/10.1145/2989225.2989232

[19] S. Nanz and C. A. Furia, "A comparative study of programming languages in rosetta code," in *Proceedings of the 37th IEEE International Conference on Software Engineering*, 2015. [Online]. Available: https://doi.org/10.1109/ICSE.2015.90

[20] R. Pereira, M. Couto, F. Ribeiro, R. Rua, J. Cunha, J. P. Fernandes, and J. Saraiva, "Ranking programming languages by energy efficiency," *Science of Computer Programming*, vol. 205, p. 102609, 2021. [Online]. Available: https://doi.org/10.1016/j.scico.2021.102609

[21] N. Van Kempen, H.-J. Kwon, D. T. Nguyen, and E. D. Berger, "It's not easy being green: On the energy efficiency of programming languages," *arXiv preprint arXiv:2410.05460*, 2024. [Online]. Available: https://arxiv.org/abs/2410.05460

[22] W. Oliveira Jr., R. Oliveira, and F. Castor, "A study on the energy consumption of android app development approaches," in *Proceedings of the 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, May 2017.

[23] A. F. d. Silva, H. N. Braganca, F. M. Q. Pereira *et al.*, "How programming languages and paradigms affect performance and energy in multithreaded applications," in *2016 IEEE 24th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*. IEEE, 2016, pp. 270–277. [Online]. Available: https://doi.org/10.1109/MASCOTS.2016.36

[24] G. Pinto, F. Castor, and Y. D. Liu, "Understanding energy behaviors of thread management constructs," in *Proceedings of the 2014 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*. ACM, October 2014, pp. 345–360.

[25] H. Lei, T. Zhang, Y. Liu *et al.*, "A green energy-efficient scheduling algorithm using the dvfs technique for cloud datacenters," in *Future Information and Communication Technologies for Ubiquitous HealthCare*, 2013, pp. 189–196. [Online]. Available: https://doi.org/10.1016/j.future.2013.06.013

[26] P. Stolf, J.-M. Pierson, and G. Da Costa, "Green it scheduling for data center powered with renewable energy," *Future Generation Computer Systems*, vol. 86, pp. 99–120, 2017. [Online]. Available: https://doi.org/10.1016/j.future.2018.03.049

[27] S. Georgiou, S. Rizou, and D. Spinellis, "Software development lifecycle for energy efficiency: Techniques and tools," *ACM Computing Surveys*, vol. 52, no. 4, pp. 1–33, 2019. [Online]. Available: https://doi.org/10.1145/3337773

[28] G. Procaccianti, H. Fernández, and P. Lago, "Empirical evaluation of two best practices for energy-efficient software development," *Journal of Systems and Software*, vol. 117, pp. 185–198, 2016. [Online]. Available: https://doi.org/10.1016/j.jss.2016.02.035

[29] C. Oduah, "Comparing rust vs. zig: Performance, safety, and more," 2024, accessed: 2025-09-19. [Online]. Available: https://blog.logrocket.com/comparing-rust-vs-zig-performance-safety-more/

[30] B. S. Sundar, ""blazingly fast" web services with zig lang using the zap library," 2024, accessed: 2025-09-19. [Online]. Available: https://medium.com/@shyamsundarb/blazingly-fast-web-services-with-zig-lang-using-the-zap-library-d64cf9e6129b