It's Not Easy Being Green: On the Energy Efficiency of Programming Languages

Nicolas van Kempen
University of Massachusetts Amherst, USA
nvankempen@cs.umass.edu

Hyuk-Je Kwon*
Unaffiliated, USA
hyukjekwon@gmail.com

Dung Tuan Nguyen*
Eureka Robotics, Vietnam
ntddebugger@gmail.com

Emery D. Berger*
University of Massachusetts Amherst, USA
Amazon Web Services, USA
emery@cs.umass.edu

***Abstract*—Does the choice of programming language affect energy consumption? Previous highly visible studies have established associations between certain programming languages and energy consumption. A causal misinterpretation of this work has led academics and industry leaders to use or support certain languages based on their claimed impact on energy consumption. This paper tackles this causal question directly: it develops a detailed causal model capturing the complex relationship between programming language choice and energy consumption. This model identifies and incorporates several critical but previously overlooked factors that affect energy usage. These factors, such as distinguishing programming languages from their implementations, the impact of the application implementations themselves, the number of active cores, and memory activity, can significantly skew energy consumption measurements if not accounted for. We show—via empirical experiments, improved methodology, and careful examination of anomalies—that when these factors are controlled for, notable discrepancies in prior work vanish. Our analysis suggests that the choice of programming language implementation has no significant impact on energy consumption beyond execution time.**

I. Introduction

The acceleration of climate change due to use of fossil fuels has driven an increased focus on efforts to decrease both the energy consumption and carbon footprint of computer systems [1–3]. In 2018, an estimated 1% of total global energy consumption was attributed to datacenters alone [4]. Modern machine learning workloads—especially training—can generate hundreds of tons of CO_2 emissions [5, 6]. For example, Meta reports that training the Llama2 large language model generated an estimated 539 tons of CO_2 emissions [7].

Programmers care about energy when building applications, but often lack tools to effectively address this concern [8]. According to an influential line of work, one potential way to reduce energy consumption is to choose a different programming language. This work analyzes a wide selection of programming languages and workloads, and concludes that different programming languages consume widely varying amounts of energy [9–11]. The centerpiece of these papers is a ranking of programming languages by energy efficiency (reproduced in part in Table I).

*Work done at the University of Massachusetts Amherst.

These studies have received wide attention, both in academic and industrial circles. They have been collectively cited over 800 times per Google Scholar (as of October 2025). The results—especially the ranking of programming languages—have had an unusually visible impact in industry, and are routinely quoted on social networks and in blog posts. As Figure 2 illustrates, the rankings have been cited by executives and engineers from Amazon [12–15], Intel [16], SAP [17], and other companies [18] to argue for business decisions and to advocate for a shift in programming languages with an eye towards sustainability. Often, these rankings have been harnessed to support the adoption of Rust, which ranks as one of the most energy-efficient languages while providing safety guarantees that languages like C and C++ lack.

Despite the fact that these studies are statistical and only establish associations, they have nonetheless been broadly interpreted as establishing a *causal* relationship, that the choice of programming language has a direct effect on a system's energy consumption. This misinterpretation stems in part from the work's presentation, not only in ranking of languages by efficiency, but also from the specific claim that "it is almost always possible to choose the best language" when considering execution time and energy consumption [11, §3.3].

The above analysis and approach suffer from numerous other methodological flaws (§II-B, §V, §VI-C). However, the primary focus of this paper is carefully addressing the question: ***does the choice of programming language affect energy consumption?*** We embrace this question by developing a rich causal model of the relationship between programming language and energy consumption. Figure 1 presents a causal diagram representing our final model. Causal diagrams illustrate how different factors influence each other, with arrows showing the direction of these influences [19]. We build this model incrementally, incorporating the various factors at play and their relationships, and provide quantitative and/or qualitative evidence supporting each step.

Our model specifically identifies the number of active cores and memory activity as the key contributors to power draw and consequently energy consumption. It highlights the importance of both programming language and application

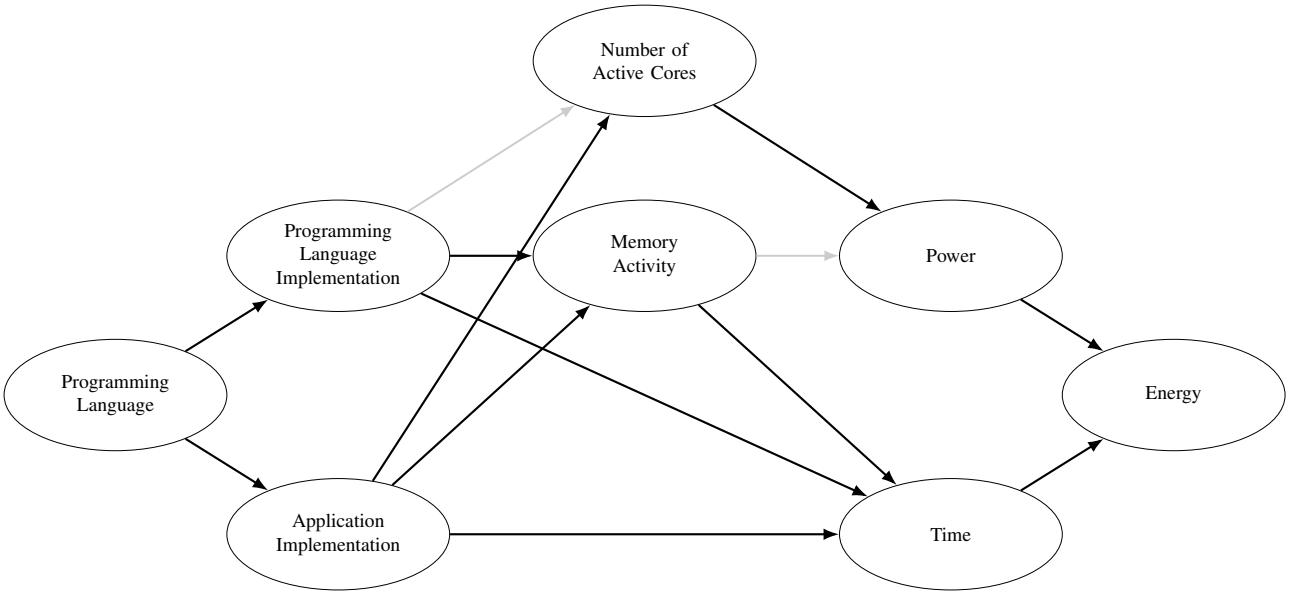
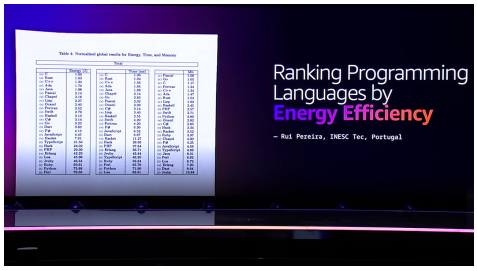


Fig. 1: **The causal model (represented as a causal diagram) of the relationship between programming language and energy consumption presented in this paper (§IV)**. Gray arrows represent comparatively weaker relationships, as Section IV details: programming language implementation has only a minor effect on parallelism, and memory activity plays a minimal, less controllable role in energy consumption compared to CPU activity.



(a) Keynote talk at AWS re:Invent 2023: "There is no reason why you should not be programming in Rust, if you are considering cost and sustainability to be high priorities".

(b) Keynote talk at KubeCon + CloudNativeCon Europe 2022: "Coding with Python over Rust for large scale applications can mean a difference of up to 75× in energy usage".

Fig. 2: **Industry keynotes advocating for Rust over other languages due to reduced energy consumption, based on the results of Pereira et al. [9–11] (§II-A).**

implementations: language implementation decisions can add execution time overheads including garbage collection or just-in-time warmup, while application implementations dictate the level of parallelism and overall performance of the program. These varying implementation characteristics are the primary drivers of the anomalies we identify in prior work.

Our experimental approach incorporates several methodological improvements, correcting technical errors in prior work that led to negative or otherwise incorrect energy readings. We present an improved methodology based on hardware performance counter data that accurately quantifies average core usage and memory activity. When controlling for confounds like varying CPU utilization, we conclude that energy consumed is directly proportional to execution time, and independent of the choice of programming language. These results simplify the task of programmers who are seeking to achieve higher energy efficiency by focusing their efforts on minimizing execution time.

II. Prior Work

A. Overview

The line of work that is the point of departure for this paper explores the relationship between the choice of programming language and energy efficiency [9–11]. These papers rank programming languages based on energy consumption, execution time, and memory usage, and attempt to find associations between these metrics. In the remainder of this paper, we refer to these papers collectively as "Pereira et al.".

In essence, these studies compare the execution time, energy consumption and memory usage of benchmark implementations in 27 various programming languages. Table I presents

TABLE I: **Partial results from Pereira et al. (§II-A)**, showing average execution time and energy consumption for the languages discussed in this paper, normalized to C. **Boldface** denotes anomalous results that Sections II-B, VI-C address.

Language	*Execution Time*	*Energy Consumption*
C	1.00	1.00
Rust	1.04	1.03
C++	**1.56**	**1.34**
Java	**1.89**	**1.98**
Go	**2.83**	**3.23**
C#	**3.14**	**3.14**
JavaScript	6.52	4.45
PHP	27.64	29.30
TypeScript	**46.20**	**21.50**
Python	71.90	75.88
Lua	82.91	**45.98**

some of the main results from Pereira et al., limited to the languages discussed in this paper, with anomalous results highlighted in boldface (Section II-B discusses this selection in detail). Their experiments leverage Intel's Running Average Power Limit (RAPL) interface to measure energy consumption, and GNU `time` or Python's `memory_profiler` to measure peak or "total" memory usage, respectively.

The programs used for comparison are from the Computer Language Benchmark Game [20] (CLBG), a corpus of small benchmark implementations in various languages (22 to 287 lines of code). The most recent paper in the series [11] also incorporates 9 benchmarks from Rosetta Code, a repository of even smaller and simpler code snippets such as Fibonacci, Ackermann, or Sieve of Eratosthenes (typically under 50 lines of code). Because Rosetta Code benchmarks are strictly smaller in scope and size than the CLBG benchmarks, we exclude them from consideration in this paper.

These studies first identify a strong relationship between energy and time. By definition, energy is a linear function of time (energy = power $\times$ time), hence a strong correlation is to be expected. They next investigate whether a fast language is always more energy efficient, and claim that this is not the case. They report no significant correlation between peak memory usage and energy consumption, but a strong correlation when considering total memory usage. Finally, they conclude by presenting a ranking of programming languages by energy efficiency, execution time, and memory usage, which programmers can use to select a language for their project.

B. Critique

As summarized here and discussed in the following sections, Pereira et al.'s studies suffer from several flaws, which this paper addresses and corrects.

Programming Language versus Implementation: Programming languages define the syntax and semantics, but it is their implementations that primarily influence performance. While some languages have only a single, widely-used implementation such as Rust or Go, others have multiple implementations, each with their own performance characteristics. For instance, Pereira et al. treat Ruby and JRuby as different languages, while they are in fact two separate implementations of the same Ruby language. The papers use different benchmark implementations to compare these two Ruby implementations, confounding their comparison.

Widely Varying Benchmark Implementations: While benchmark implementations are claimed to employ the "exact same algorithm" across languages [11, §1], this is in fact not the case. Benchmark implementations notably have highly varied levels of parallelism and CPU usage, varying degrees of use of third-party libraries, and non-uniform use of vector instructions. These important differences are not properties of the languages themselves, and their effect on performance and energy consumption must be accounted for.

Apparent Anomalies in Results: Some results reported by Pereira et al. are counter-intuitive, and are presented without investigation or explanation. C++ is reported as being 34% less energy efficient and 56% slower than C. Since C++ is approximately a superset of C, and both share the same compiler, optimizations, and code generation backend, we would expect identical energy consumption and execution time. Similarly, TypeScript is reported as being $4.8\times$ less energy efficient and $7.1\times$ slower than JavaScript. TypeScript is a strict superset of JavaScript: any valid JavaScript program is a valid TypeScript program. The same Node.js runtime system is used for both languages. The compilation process for TypeScript may insert or modify code to support older JavaScript standards, but we do not expect this to result in any significant performance overhead. Java, C#, and Go are reported as $1.89\times$, $3.14\times$, and $2.83\times$ slower than C, respectively. These numbers are unexpectedly high as these implementations are known to be highly optimized [21]. We expect low overhead from garbage collection costs, plus initial just-in-time compilation overhead for Java and C#. Lua and TypeScript both stand out as exhibiting significantly lower normalized energy consumption than normalized execution time; this is explained by the fact that nearly all of the Lua and TypeScript benchmark implementations are sequential, while the implementations of these benchmarks in other languages are parallelized. Section VI-B characterizes the impact of the number of active cores on energy consumption.

Inappropriate Memory Metrics: Peak memory usage and "total" memory usage are used to estimate memory activity. The tools used to gather both metrics are both based on resident set size (RSS), which is a poor proxy for memory usage and activity [22]. RSS includes memory that may not be actively used by the program, and crucially does not take cache activity into account. A well-optimized program can have a large RSS but excellent cache locality: if the cache can fulfill most memory requests, memory activity will be low. Benchmark implementation specifics heavily influence peak memory usage, which depends primarily on the choice of data structures used throughout the program. These differences should not be attributed to the languages themselves.

The total memory usage is measured by using Python's `memory_profiler`, which samples the RSS at a frequency of ten times per second. This metric is directly proportional to

execution time, which is itself directly proportional to energy consumption.

Failure to Control for Language Implementation Characteristics: Most language implementations have an initial cost to import and set up necessary in-memory data structures. Language implementations using a just-in-time compiler also require an initial warmup period [23, 24]. These start-up costs are amplified by the benchmarks' short execution times, which in some cases run for less than a fraction of a second. For example, measuring only the first iteration of a short-lived Java benchmark may not be indicative of Java's overall performance. Section VI-C shows that this first iteration can be up to 3× slower than subsequent ones. Garbage collection also can have a significant impact on both execution time and memory usage [25], and can be fine-tuned to obtain better performance.

III. Research Questions

This paper provides numerous improvements to methodology and results over prior work. Section IV first introduces and gradually builds a causal model capturing all factors standing between choice of programming language and energy consumption. Section V describes the improved methodology used in this paper, including correction of technical errors and enhanced data gathering with performance counters. Finally, using both the causal model and improved methodology, Section VI addresses the following research questions:

- **RQ1**: Do some language implementations consume more energy than others?
- **RQ2**: What are the key contributors to power draw standing between choice of programming language and energy consumption?
- **RQ3**: Can observed anomalies in prior work be explained through the lens of our causal model?

IV. Causal Model for Energy Consumption in Programming Languages

This section builds piece by piece the causal model presented as a diagram in its final form in Figure 1. Causal diagrams illustrate how different factors influence each other, with directed edges indicating cause-to-effect relationships [19]. Formally, edges indicate that changes in the origin node impact the probability distribution of its descendants. Our diagrams additionally use gray arrows to represent comparatively weaker relationships, as detailed in the following sections. Each subsection below gradually adds nodes and edges to the diagram in order to obtain a rich causal model that captures the essential factors in the relationship between the choice of programming language and energy consumption. Figures 3, 4, 5, and 6 build upon each other, with new elements in orange. They culminate in the final model represented in Figure 1.

A. Starting Point

Our starting point, shown in Figure 3, is that the choice of programming language has a direct effect on energy consumption. The following sections will explore additional elements that come into play in this relationship, and build up a complex model exploring the impact of different factors on energy consumption.

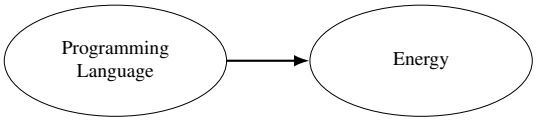


Fig. 3: **The simple model used as a starting point in this paper (§IV-A).** This simple model captures the relationship implied in Pereira et al., namely that the choice of programming language has a direct impact on total energy consumption.

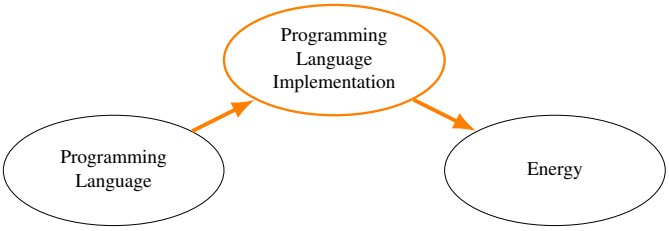


Fig. 4: **Programming languages may have multiple implementations (§IV-B)**, and implementation decisions such as garbage collection or using just-in-time compilation have a more direct impact on performance. We are in fact comparing implementations of programming languages, not the languages themselves.

B. Programming Languages and Implementations

Figure 4 introduces the important distinction between the programming languages themselves and their implementations. Some languages have a single well-known implementation, or a blurry line between language and implementation, such as Rust or Go. Others have multiple implementations, such as C/C++ with GCC, Clang, or MSVC, Java with various JVMs, Python with CPython and PyPy, or Lua with LuaJIT. Finally, there can be significant differences between specific *versions* of the same implementation. Our model treats those different versions as different implementations altogether. For example, recent CPython releases each successively delivered significant performance improvements, notably with the 3.11 release achieving a 25% speedup over 3.10 [26].

Of course, programming languages have an effect on implementation possibilities. They may dictate the general memory layout of objects, the need for a garbage collector, or the possibility of ahead-of-time compilation. For example, Java has no alternative for memory reclamation other than a garbage collector. Other dynamic features such as dynamic typing, reflection, or `eval` further limit the range of options available to implementers.

The impact of the characteristics of programming languages on their implementations is transitively included in our model: the language has a causal influence on the implementation, which in turn has a direct causal impact on execution time.

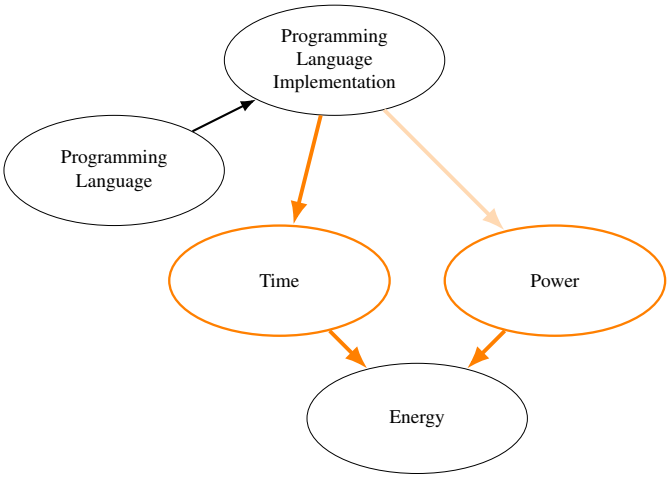


Fig. 5: **Energy consumption is the product of power and time (§IV-C).** Splitting into these two components allows separate consideration of each factor.

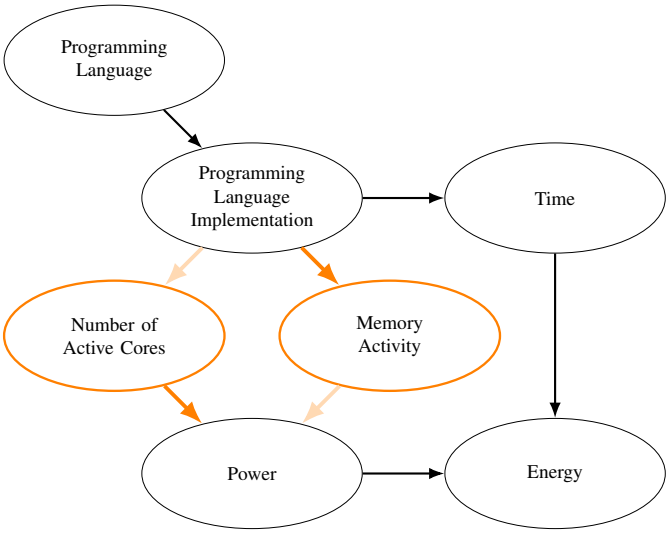


Fig. 6: **Number of active cores is the primary factor in increased power draw (§IV-D).** Memory activity also increases power draw, but to a much lesser extent. Figure 8 quantifies this.

The performance and energy consumption impact of those language features is mediated by the implementation. Dynamic dispatch, for example, may be implemented or optimized with v-tables, fat pointers, monomorphic and polymorphic inline caches, or profile-guided optimization.

C. Decomposing Energy into Power and Time

Fundamentally, energy consumption is the product of power and time. Figure 5 decomposes these two factors. Section VI-B demonstrates that power draw is constant across all languages, implementations, and benchmarks. Claims of the contrary in prior work were in large part due to missing multicore accounting, directly comparing sequential and concurrent benchmarks.

In theory, runtime systems running in parallel to the user program may affect energy consumption beyond variance in execution time, since increased core usage will increase power draw, as Section VI-B details below. This hidden increased parallelism could occur with a parallel garbage collector or just-in-time compilation threads executing alongside the main program. In practice, compilation threads are primarily active only during startup, and both are rarely significant compared to the main program's execution and other general overhead costs. For this reason, we draw this edge in gray in our model.

D. Contributors to Power Draw

Previous research has demonstrated that processor and memory energy usage are the main variable contributors to power draw [27]. We add these two factors to our model in Figure 6. For the studied programs on our experimental platform, the ratio of memory energy consumption to CPU energy consumption remains between 2 and 8% (Section VI-B). Further, memory activity is not something that is easily controlled by the programmer. While certain algorithms or data structures maximize cache locality, there is eventually a limit stemming from the LLC's fixed, relatively small size. Programming languages or their implementations may impose additional overhead with a garbage collector, memory allocation overhead, or object memory layout. Because of the difficulty of controlling memory activity, as well as its comparatively smaller impact compared to CPU activity on power draw, we draw this edge in gray in our model.

E. Application Implementations

We add a final node to our diagram: the application implementations themselves, which have the most direct impact on energy consumption. Figure 1 presents the final causal model developed in this paper. Application implementation decisions dictate the level of parallelism, the algorithms used, and the data structures employed.

F. Other Factors

There are many other factors that may affect energy consumption. CPU voltage and frequency scaling has a direct impact on power consumption, and there is active research focusing on reducing or increasing frequency based on memory activity or other factors to draw less power for a comparatively smaller performance penalty [28]. Specific processor model and architecture also have an impact on power draw, with newer processors generally being more energy-efficient. These factors are unrelated to choice of programming languages and are controlled for in our experiments.

V. Enhanced Methodology

Our methodology improves upon prior work in several ways. First, it corrects technical errors in the measurement tool used by Pereira et al. [29]. It also enhances data gathering by collecting performance counter readings, which we use to identify key contributors to energy consumption. Finally, it expands the set of benchmarks used to include two additional widely-used benchmark suites.

A. Measurement Tool

We develop a new measurement tool for this study, correcting errors found in Pereira et al.'s implementation [29], and adding performance counter readings capturing the number of cores used and memory activity. The tool reads performance counters at the beginning and end of each benchmark run, and samples energy consumption readings once per second. Periodic sampling is necessary to ensure correct energy readings, as the next section explains.

RAPL: Starting with the Sandy Bridge microarchitecture (2011), most Intel processors provide a power management interface called Running Average Power Limit (RAPL). RAPL allows measurement of the energy consumption of various parts of the system [30]. The interface is exposed via Model Specific Registers (MSRs), of which the following measure energy consumption:

- `MSR_RAPL_POWER_UNIT` contains information used to convert the raw energy status counter value to joules.
- `MSR_PKG_ENERGY_STATUS` contains the raw energy status counter for the entire processor package (denoted PKG in our graphs).
- `MSR_DRAM_ENERGY_STATUS` contains the raw energy status counter for the random access memory attached to that processor.

MSRs track energy consumption at the granularity of an entire package. There is no way to assign energy consumption to a specific thread or process, but only to the system as a whole. In particular, there is no way to eliminate overhead from the measurement tool and other background processes from the measurement. The measurement tool must therefore be as lightweight as possible. We disable all non-essential processes, keeping only those necessary for the machine to operate, and assume these processes remain constant across all experiments.

Further, RAPL samples include all cores, even if the program under test only uses a single core. If a benchmark is single-threaded or generally uses fewer cores than available, idle cores will be included in the energy consumption measurement. Therefore, using a varying level of parallelism across benchmark implementations can result in unfair comparison, as idle cores will add some constant energy consumption to each sample.

An inspection of Pereira et al.'s data files [29] reveals the presence of multiple negative energy readings. Of course, negative energy consumption is not physically possible. We expect that these values were removed by the outlier removal process described in their work and did not affect final results. To ensure correct energy readings, we rectify the following errors in the tool used by previous work:

- **Erroneous inclusion of upper bits of status counters:** The previous tool incorrectly includes the upper 32 bits of the energy status counters in their accounting. These upper bits are reserved and must be discarded; our tool tracks only the bottom 32 bits, which contain the actual count of consumed energy.
- **Failure to prevent overflow of status counters:** The previous tool reads only a start and end value of the energy status counter, but that counter overflows after "around 60 seconds when power consumption is high" [31]. To avoid overflow for long-running computations, our tool uses a separate thread that periodically (at 1Hz) reads the contents of this register.
- **Erroneous floating point arithmetic:** Finally, the previous tool incorrectly subtracts counter readings after scaling and conversion to floating point. This conversion will result in negative or otherwise incorrect results when overflow occurs. Our tool avoids this problem by converting results to floating point only after subtracting consecutive readings.

Performance Counters: Hardware performance counters are a feature of modern processors that track various events, such as total cycles, cache misses, branch misses, and so on. They are often used to analyze program performance, notably when profiling. These counters are 64 bits in length and thus not susceptible to overflow, so it suffices to read them once at the beginning and once at the end of each experiment.

Memory Activity: Section IV-D highlights the impact of memory activity on energy consumption: a majority of memory energy consumption is dependent on the rate of read and write operations. We monitor the last level cache (LLC) using performance counters to estimate memory activity. Each LLC hit means that the cache already had a copy of the data, and therefore no further memory activity occurs. On the other hand, each LLC miss means that the system must fetch data from memory. Therefore, the number of LLC misses is a good proxy for memory activity.

Average CPU Usage: There are multiple ways to measure average core usage while a program is running. For simplicity, we leverage performance counters provided by the operating system. The Linux kernel exposes the `task-clock` software event counter, which aggregates in nanoseconds the time spent on all processor cores. To obtain average core usage, we divide this counter's value by the total execution time of the program.

B. Benchmarks

This paper extends the evaluation performed by the original Pereira et al. study by incorporating two additional large-scale benchmark suites. We first include all possible benchmark implementations from the Computer Language Benchmark Game used by Pereira et al. that successfully compile and run (118 benchmarks). Using the same set of benchmarks allows reproduction and re-examination of results in Section VI-C. In addition, the analysis in Sections VI-A and VI-B includes two more widely-used benchmark suites: SPEC CPU 2017 [32] (18 benchmarks) and DaCapo Chopin [33] (22 benchmarks). The following subsections provide a brief overview of each of these benchmark suites.

Computer Language Benchmark Game: The Computer Language Benchmark Game [20] (CLBG) is a collection of small programs implemented in many different programming languages (22 to 287 lines of code). We use the same

TABLE II: **Computer Language Benchmark Game benchmarks (§V-B)**, along with the range of lines of code (LoC) across languages for each benchmark as reported by `cloc`.

Benchmark	*LoC*	*Description*
binary-trees	36–98	Memory allocator stresstest
fannkuch-redux	40–158	Permutation algorithm
fasta	84–287	Random DNA generation
k-nucleotide	57–202	Frequency counting
mandelbrot	32–140	Fractal rendering
n-body	78–157	Physics simulation
pidigits	41–149	Arbitrary precision
regex-redux	22–111	String processing
reverse-complement	34–257	String transformation
spectral-norm	33–156	Linear algebra

benchmark implementations as Pereira et al. [29], which is in effect a snapshot of the fastest versions of the benchmarks as available on the CLBG repository at the time of the original work. Table II provides a brief description of each benchmark. We limit our analysis to eleven languages across thirteen implementations: C, C#, C++, Go, Java, JavaScript, Lua (Lua and LuaJIT), PHP, Python (CPython and PyPy), Rust, and TypeScript. These languages comprise 11 of the top 12 most popular general programming languages in the 2024 Stack Overflow survey [34] (excluding Kotlin).

We attempt to compile and run all benchmark implementations with sources present in the repository, and omit those without source code or which fail with compilation or runtime errors. After these omissions, the analysis below spans 118 out of 130 possible language implementation / benchmark pairs. Some benchmarks make use of third-party libraries: we use the default package manager's version when available, or manually upgrade to the latest available version.

We stress that while the CLBG benchmarks are not necessarily representative of real-world applications, the causal analysis this paper develops is largely independent of the details of the benchmark implementations. It instead highlights the impact of high-level *properties* of the benchmark implementations, such as their degree of parallelism and cache activity.

SPEC CPU 2017: SPEC CPU 2017 [32] is a widely used benchmark suite for evaluating systems on C, C++ and Fortran code. The suite consists of large real-world programs and workloads, ranging between 1k and 1.3M lines of code. We use the latest available version (1.1.9) and run all benchmarks in the "speed" category using the reference input set. We omit a single benchmark, cam4, as it did not successfully run on our platform. Table III briefly describes each benchmark.

DaCapo: DaCapo [33] is a widely used benchmark suite for Java applications. The suite consists of large real-world programs and workloads, ranging between 25k and 6M lines of code. We use the latest available version (Chopin) and run all benchmarks using the default input set. Table IV provides a brief description of each benchmark.

TABLE III: **SPEC CPU 2017 benchmarks (§V-B)**.

Benchmark	*Language(s)*	*Description*
perlbench	C	Perl interpreter
gcc	C	GNU C compiler
mcf	C	Route planning
omnetpp	C++	Discrete-event simulation
xalancbmk	C++	XML to HTML conversion
x264	C	Video compression
deepsjeng	C++	Alpha-beta tree search
leela	C++	Monte Carlo tree search
exchange2	Fortran	Recursive solution generator
xz	C	General data compression
bwaves	Fortran	Explosion modeling
cactuBSSN	C++, C, Fortran	Physics: relativity
lbm	C	Fluid dynamics
wrf	Fortran, C	Weather forecasting
cam4	Fortran, C	Atmosphere modeling
pop2	Fortran, C	Wide-scale ocean modeling
imagick	C	Image manipulation
nab	C	Molecular dynamics
fotonik3d	Fortran	Computational electromagnetics
roms	Fortran	Regional ocean modeling

TABLE IV: **DaCapo benchmarks (§V-B)**.

Benchmark	*Description*
avrora	AVR controller simulation
batik	Render SVG files
biojava	Protein sequence analysis
cassandra	NoSQL database
eclipse	IDE workload
fop	Render PDF files
graphchi	ALS matrix factorization
h2	SQL database
h2o	Machine learning
jme	Render video game frames
jython	Python implementation
kafka	Stream processing
luindex	Text indexing
lusearch	Text search
pmd	Static analysis
spring	Web framework
sunflow	Render photorealistic images
tomcat	Web server
tradebeans	Stock market simulation
tradesoap	Stock market simulation
xalan	XML transformation
zxing	Barcode reader

VI. Evaluation

This section addresses the research questions posed in Section III, using the causal model and enhanced methodology introduced in Sections IV and V.

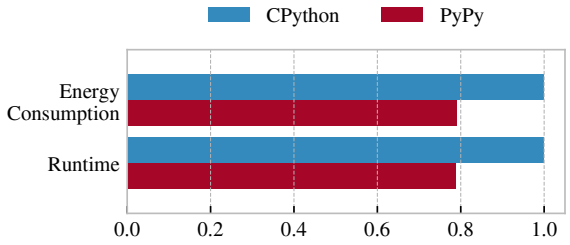
We conduct experiments in this paper using a server equipped with two Intel Xeon Gold 6430 processors totaling 128 logical cores and 128GB of memory, running Linux version 6.8.0-45-generic. Table V details specific compiler and runtime system versions. Benchmark runs are isolated in Docker containers to ensure reproducibility. Docker introduces negligible performance overhead [35], which we confirm for the experiments detailed in this paper by obtaining equivalent results without Docker. Docker provides easy access to control groups, making it possible to place restrictions on CPU usage or to pin programs to execute on specific cores (Section VI-B).

A. *RQ1: Do Some Language Implementations Consume More Energy Than Others?*

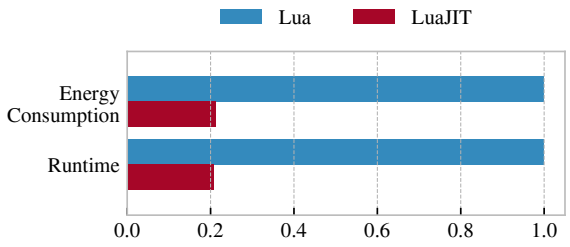
According to our model, the only significant causal path from programming language to energy consumption is me-

TABLE V: **Versions of compilers and runtime systems used (§VI).** All are most recent as of September 2024.

Implementation	*Language(s)*	*Version*
LLVM / Clang	C, C++	19
Rust	Rust	1.81.0
OpenJDK	Java	21.0.4+7
Go	Go	1.23.1
C# / .NET	C#	8.0.8
Node.js	JavaScript, TypeScript	20.17.0
PHP	PHP	8.3.11
tsc	TypeScript	5.6.2
CPython	Python	3.12.6
PyPy	Python	7.3.17
Lua	Lua	5.4.7
LuaJIT	Lua	87ae18a



(a) CPython versus PyPy



(b) Lua versus LuaJIT

Fig. 7: **Programming Languages can exhibit different performance characteristics depending on the implementation used (§IV-B).** Here, we compare interpreters (CPython, Lua) to their JIT equivalent (PyPy, LuaJIT) on the same CLBG benchmark sources. We normalize to the interpreter results. We find around $5\times$ and $1.25\times$ decreased execution time and energy consumption on average for LuaJIT over Lua and PyPy over CPython, respectively.

diated by three factors: the language's implementation, the application's implementation, and execution time. Therefore, for identical application implementations, power draw variance can only be due to language implementation differences. Similarly, if the number of cores used is fixed, variance in energy consumption is uniquely determined by execution time.

For the Same Language: It is well known that different implementations of the same language may have widely varying performance characteristics. Figure 7 compares languages across multiple implementations on the same set of benchmarks, namely CPython/PyPy and Lua/LuaJIT, and confirms that not only do the JIT implementations offer increased performance, they also provide corresponding energy savings. On average for these benchmarks, PyPy is about $1.25\times$ faster than CPython, and LuaJIT is $5\times$ faster than the Lua interpreter.

Note that different implementations of the same language, as well as different versions of the same implementation, may have different tooling, compatibility, or support. These differences do not affect our experiments. In our PyPy/CPython and Lua/LuaJIT evaluation, all benchmarks run successfully in both the interpreter and JIT implementation, with only one exception for each: pidigits does not work with PyPy due to a third-party library compilation failure, and fasta fails in LuaJIT due to a standard library utility rename.

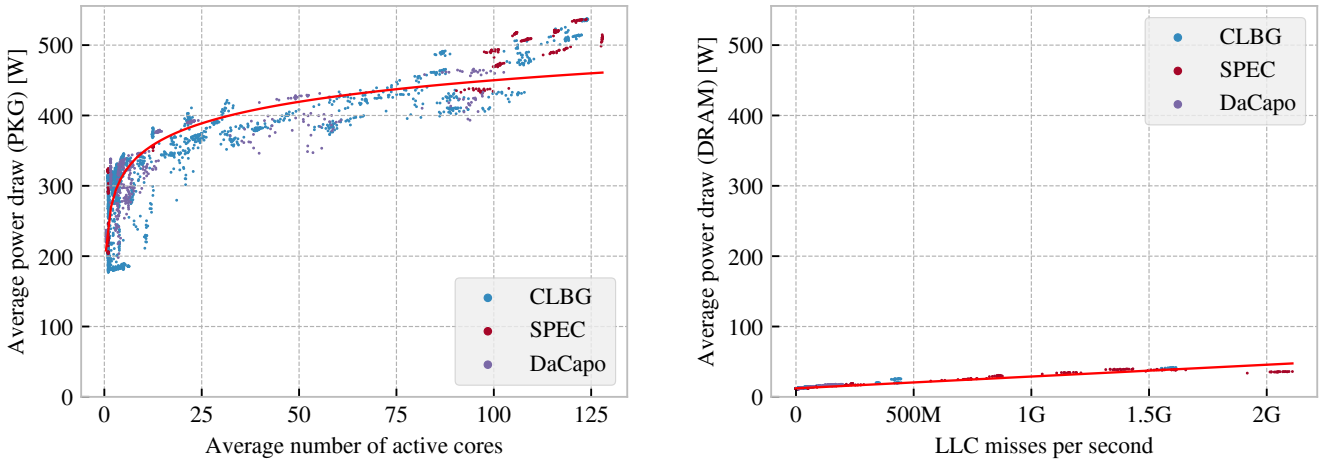
Across Many Languages and Implementations: To test if the choice of programming language implementation has an effect on power draw beyond execution time, we fix other factors that may affect power draw: we limit the program to a single core via Docker's `--cpuset-cpus` argument, and pin CPUs to their minimum frequency with turbo mode disabled to ensure frequency scaling and throttling do not increase power draw variance in our measurements. Once these factors are controlled for, we measure execution time and energy consumption of each benchmark across all language implementations. These measurements yield equal power draw: 189.8 ± 0.5 W. Hence, the impact of choice of programming language implementation (or programming language more generally) on energy consumption beyond execution time is negligible.

RQ1 Summary: Different language implementations can have widely varying performance characteristics, and by extension, energy efficiency. However, once accounting for external factors such as numbers of cores used, all benchmarks in all language implementations exhibit constant power draw. Hence, energy consumption is directly proportional to execution time. Faster language implementations will offer commensurate energy savings.

B. *RQ2: What Are the Key Contributors to Power Draw Standing Between Choice of Programming Language and Energy Consumption?*

Our causal model identifies number of active cores and memory activity as primary contributors to power draw. Section V details our improved measurement methodology, which allows us to accurately quantify to which extent these factors actually contribute to power draw.

Number of Active Cores: Figure 8a shows the relationship between the number of active cores and power draw. Using multiple cores, energy usage grows up to around $2\times$ the base power draw compared to using a single core. We fit a log curve to the data $\left(R^2 = 0.72\right)$, as the relationship does not appear linear, but results may vary across platforms. Denoting $\text{Power}\,(x)$ as the average power draw using x cores, we



(a) **Average package (PKG) power draw by average number of active cores.** Regression: $y = 31 \log_2 x + 246$ $(R^2 = 0.72)$.

(b) **Average DRAM power draw by memory activity**, approximated using LLC misses. Regression: $y = 1.68 \cdot 10^{-8} x + 12$ $(R^2 = 0.93)$.

Fig. 8: **Power draw is much more significantly affected by the number of active cores than memory activity (§IV-D).** Each point on those graphs represents a single benchmark run.

find that, on our system, doubling the number of cores used increases average power draw by only roughly 31W:

$$\begin{aligned} &\text{Power}\,(x) = 31 \log_2 x + 246 \\ \Rightarrow\ &\text{Power}\,(2x) = 31 \log_2 (2x) + 246 \\ \Rightarrow\ &\text{Power}\,(2x) = 31 \log_2 2 + 31 \log_2 x + 246 \\ \Rightarrow\ &\text{Power}\,(2x) = \text{Power}\,(x) + 31 \end{aligned}$$

Further, on our machine, doubling the number of cores increases relative energy efficiency, provided throughput increases by at least 13%:

$$\frac{\text{Power}\,(2x)}{\text{Power}\,(x)} = \frac{31 \log_2 (2x) + 246}{31 \log_2 x + 246} \leq 113\% \quad (1 \leq x \leq 128)$$

In other words, parallelization overhead up to 87% is acceptable. Therefore, on our experimental platform, aggressively parallelizing programs is nearly always an energy-efficient choice.

Memory Activity: Previous research has established that roughly 40% of dynamic random access memory (DRAM) energy consumption is constant and required to refresh memory cells to keep the system running properly, while the remaining 60% is related to read and write activity [36]. Section V-A introduces last level cache (LLC) misses as a better proxy for memory activity: any read or write request that cannot be fulfilled by the LLC will have to go to DRAM. Figure 8b shows a strong linear correlation between LLC misses and increased power draw $(R^2 = 0.93)$. However, this figure also highlights how much smaller the memory's energy consumption is to the processor's, with their ratio remaining from 2 to 8 % across all benchmarks.

RQ2 Summary: Number of active cores is the primary factor in increased power draw. Memory energy consumption is linearly related to memory usage, but significantly less important than processor energy consumption. Moreover, aggressive parallelization, even with high overheads, increases energy efficiency.

*C. **RQ3**: Can Observed Anomalies in Prior Work Be Explained Through the Lens of Our Causal Model?*

This section identifies four major sources of anomalies in prior work. For each anomaly, we indicate in brackets which causal path(s) in our model are responsible.

Concurrency [Application Implementation → {Number of Active Cores, Time}]: The number of cores used by an application is a major factor in energy consumption, as Section VI-B quantifies. Therefore, comparing benchmark implementations in different languages that use different numbers of cores is not a fair comparison. Level of parallelism imbalances are the main reason for the reported performance discrepancies between JavaScript and TypeScript. The TypeScript version of the mandelbrot benchmark is 21× slower than its JavaScript implementation. Its execution is fully sequential, while the JavaScript version uses 28 cores on average.

Further, since TypeScript is a strict superset of JavaScript and we test them using the same runtime system, there should be no differences in execution time or energy consumption. With some minor edits in four benchmarks, all JavaScript benchmark implementations pass the TypeScript compiler without any errors, and yield equal performance.

Third-Party Library Usage [Application Implementation → {Time, Number of Active Cores}]: Choice of third-party libraries used has a significant impact on application perfor-

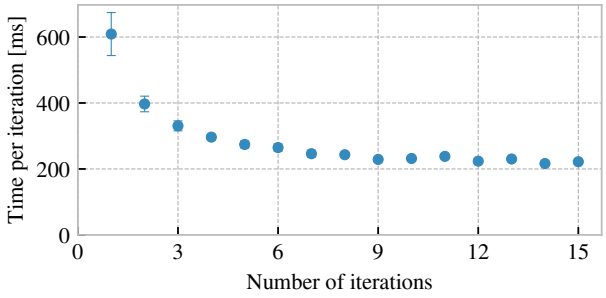
mance. In fact, regular expression benchmarks are often poor candidates to make any comparison beyond the library used. This is the case for the CLBG regex-redux benchmark: it is $8.9\times$ slower in its C++ implementation compared to the C version. This difference is entirely due to choice of third-party library: the C version uses the PCRE library, while the C++ version uses the Boost library. On this benchmark, Boost's library performs significantly worse than PCRE. This outlier alone accounts for the entire reported gap between C and C++.

Unlike TypeScript and JavaScript, C++ is not a strict superset of C. Nonetheless, all C benchmarks compile in C++ mode, only sometimes requiring the `-fpermissive` compiler flag or minor changes such as added types or casting. These programs then yield identical performance and energy consumption numbers.

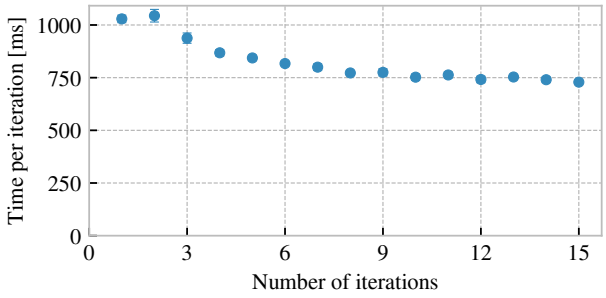
Crossing Language Boundaries: Using third-party libraries also allows for easily crossing language boundaries, for instance to access lower-level system utilities or to obtain greater performance. In Python, using NumPy can yield up to $60,000\times$ faster code when multiplying matrices [22, 37]. On a small micro-benchmark multiplying $10,000 \times 10,000$ matrices, we find that NumPy is $84\times$ faster than a naive three-loop C++ implementation, with $44\times$ lower energy consumption. Of course, this is because NumPy's underlying matrix-multiply implementation is written in a low-level language and leverages many optimizations such as parallelism, vector instructions, blocking, or hardware-specific instructions. To some degree, using NumPy and other third-party libraries has become the "correct" way of writing Python code: leveraging low-level languages for compute-intensive tasks, while keeping Python's simplicity. Even C++ may cross over to code originally written using C, Fortran, or direct Assembly for performance when using BLAS.

Just-in-Time (JIT) Warmup [Programming Language Implementation $\rightarrow$ Time]: JIT program execution is typically split into a startup phase and a delayed steady state phase of peak performance. As the program executes, the JIT runtime determines which code paths are frequently executed, and compiles them into machine code, eventually optimizing hotspots. However, existing research has already shown that a steady state of peak performance is not always reached [23, 24]. In fact, one of these studies [24] uses the same CLBG benchmark suite and demonstrates that for some benchmarks, a steady state is never reached.

Factors other than JIT warmup may also affect the first iteration of short-lived benchmarks, such as bytecode interpretation or memory and garbage collection initialization. To verify this, we wrap Java benchmarks in a loop and gradually increase the number of in-process iterations. Figure 9 shows that for the mandelbrot and fannkuch-redux benchmarks, the first iteration's execution time is respectively $2.7\times$ and $1.4\times$ slower than when averaging over 15 iterations. Generally, we see that for these and other benchmarks, averaging over ten or more iterations amortizes the first iteration's impact: across all CLBG benchmarks, we observe a mean of 80% improved energy efficiency and execution time.



(a) mandelbrot



(b) fannkuch-redux

Fig. 9: **Measuring just the first iteration is not indicative of Java's (OpenJDK) performance for long-lived applications (§IV-E).** We gradually increase the number of iterations, observing decreased per-iteration execution time on several benchmarks, including (a) mandelbrot and (b) fannkuch-redux.

Garbage Collection [Programming Language Implementation $\rightarrow$ {Time, Memory Activity}]: An important property of higher-level languages and their implementations is garbage collection, which imposes significant performance overhead [25]. Out of the 13 language implementations discussed in this paper, 10 use a garbage collector. Garbage collectors can typically be configured and tuned for each application to offer maximal performance, which is standard practice in real-world applications to optimize for application throughput and/or latency. For example, Go's garbage collector can be disabled entirely with the `GOGC` environment variable. When doing so, we observe a $2.8\times$ speedup on the allocation-intensive binary-trees benchmark, and a $1.23\times$ speedup on mandelbrot. Disabling the garbage collector over all Go benchmarks yields an average of $1.15\times$ improved execution time and energy consumption.

RQ3 Summary: Variations in application implementations, notably parallelism and third-party library usage, explain observed anomalies between JavaScript/TypeScript and C/C++. Important language implementation specifics such as warmup and garbage collection must also adequately be accounted for: when doing so, the unexpectedly large gap between Go/Java and C significantly shrinks.

VII. Threats to Validity

This study largely focuses on reproducing Pereira et al.'s experiments, which rely on a specific set of benchmarks. We mitigate this threat to external validity by not only focusing on general run-time properties of those programs rather than their absolute execution time, but also by verifying results across two additional benchmark suites.

Experiments in this paper are conducted on a server architecture, maintaining comparability with prior work. While we expect our conclusions to generalize to other architectures, results may differ on mobile, embedded systems, or architectures mixing energy-saving and high-performance cores.

VIII. Other Related Work

Previous sections, notably Section II, discuss and critique the main line of research relating programming languages and energy consumption [9–11]. Follow-on studies [38, 39] use external energy measurements rather than on-chip measurements, but still suffer from the same limitations as the original work.

Georgiou et al. [40] explore the Energy Delay Product (EDP) of programming languages on Rosetta Code. The goal of EDP is to establish a trade-off between energy consumption and execution time. Our evaluation suggests that, in fact, optimizing for execution time also provides proportional energy consumption reduction. Georgiou et al.'s public data reveals that this conclusion is in line with their findings: even in the "optimize for energy" case, nearly all benchmark versions which are the fastest are also the most energy-efficient, with the rare exceptions attributable to short execution times, coarse measurement granularity, and/or missing multicore accounting. Our methodology leverages modern on-chip energy and performance counters, offering an analysis and new causal model based on more granular and precise measurements.

Several other studies [21, 41] explore and compare performance and other properties across programming languages, but do not address energy efficiency directly. These comparative studies are complementary to this work: we show here that execution time is the primary factor in energy consumption.

Processor operating voltage and frequency have an impact on energy consumption. This impact has been studied in the context of trading off speed for reduced energy consumption, where compiling for speed yields better energy efficiency in the general case [42]. For specific applications, such as sparse matrix computations, compile-time techniques may decrease energy usage without impacting execution time by leveraging load imbalance [43]. Other work has shown that using characteristics of a program during execution to reduce the processor's voltage or frequency can yield higher energy efficiency for a comparably smaller performance degradation [44]. For example, performance counter information can be used to scale the processor's frequency [28]. These approaches are orthogonal to the choice of programming language and could be applied in many programming environments.

Graphics processing units (GPUs) are increasingly prevalent in modern computing systems, notably in artificial intelligence workloads. Past work has examined the energy efficiency of GPUs from a software perspective, notably accounting for the energy expenditure of tensors [45] and comparing the energy consumption of different machine learning frameworks [46]. Extending our work to incorporate a model for energy consumption of GPUs is a potential avenue for future work.

Previous research also addresses the energy consumption of different data structures in certain languages, notably Java collections [47–49], or Python data manipulation libraries [50]. Data structures can be interesting candidates to study through an energy-focused lens as they can have a substantial impact on locality and memory activity, which, as Section IV-D describes, are factors in energy consumption. However, programmers often have a limited choice of data structures that fit their use case or algorithm.

Finally, past work has also investigated energy efficiency of concurrent programs, focusing on Haskell's data sharing primitives [51] or Java's thread management constructs [52], or analyzing the power to performance trade-offs in lock-based synchronization [53]. As Section IV-D argues, these studies confirm that the power usage patterns of the processor and the machine as a whole are more complex with parallel execution.

IX. Conclusion

This paper presents a detailed causal model exploring the complex relationship between choice of programming language and energy consumption. It captures some of the factors at play, notably distinguishing implementation from programming language, and establishing the number of active cores and memory activity as important factors in power draw.

Using this causal model, we investigate and explain anomalies in previous research, finding that many factors such as parallelism level, benchmark implementation specifics, or language implementation properties must be taken into account for a fair comparison. Our results suggest that the choice of programming language has no significant impact on energy consumption beyond execution time. Programmers aiming to reduce energy consumption can do so by focusing on performance optimizations. This strategy is possible even in "inefficient" programming languages like Python by using faster language implementations, employing faster and more parallel algorithms, and using native libraries.

Data-Availability Statement

Data supporting this paper's evaluation is available at `github.com/nicovank/Energy-Languages`.

Acknowledgments

We thank Prashant Shenoy, Walid Hanafy, Noman Bashir, Mehmet Savasci, and Abel Souza for their help getting started with energy measurements using RAPL, their continuous feedback, and for providing access to an energy-measurement-ready machine. We thank Eunice Jun for suggesting exploration of causal models in this work.

References

[1] W. A. Hanafy, Q. Liang, N. Bashir, A. Souza, D. Irwin, and P. Shenoy, "Going Green for Less Green: Optimizing the Cost of Reducing Cloud Carbon Emissions," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ser. ASPLOS '24. New York, NY, USA: Association for Computing Machinery, 2024, pp. 479–496. [Online]. Available: https://doi.org/10.1145/3620666.3651374

[2] W. A. Hanafy, Q. Liang, N. Bashir, D. Irwin, and P. Shenoy, "CarbonScaler: Leveraging Cloud Workload Elasticity for Optimizing Carbon-Efficiency," in *Abstracts of the 2024 ACM SIGMETRICS/IFIP PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS/PERFORMANCE '24. New York, NY, USA: Association for Computing Machinery, 2024, pp. 49–50. [Online]. Available: https://doi.org/10.1145/3652963.3655048

[3] A. Radovanovic, R. Koningstein, I. Schneider, B. Chen, A. Duarte, B. Roy, D. Xiao, M. Haridasan, P. Hung, N. Care, S. Talukdar, E. Mullen, K. Smith, M. Cottman, and W. Cirne, "Carbon-Aware Computing for Datacenters," 2021. [Online]. Available: https://arxiv.org/abs/2106.11750

[4] E. Masanet, A. Shehabi, N. Lei, S. Smith, and J. Koomey, "Recalibrating Global Data Center Energy-Use Estimates," *Science*, vol. 367, no. 6481, pp. 984–986, 2020. [Online]. Available: https://www.science.org/doi/abs/10.1126/science.aba3758

[5] E. Strubell, A. Ganesh, and A. McCallum, "Energy and Policy Considerations for Deep Learning in NLP," in *Proceedings of the 57th Conference of the Association for Computational Linguistics, ACL 2019, Florence, Italy, July 28- August 2, 2019, Volume 1: Long Papers*, A. Korhonen, D. R. Traum, and L. Màrquez, Eds. Association for Computational Linguistics, 2019, pp. 3645–3650. [Online]. Available: https://doi.org/10.18653/v1/p19-1355

[6] D. A. Patterson, J. Gonzalez, Q. V. Le, C. Liang, L. Munguia, D. Rothchild, D. R. So, M. Texier, and J. Dean, "Carbon Emissions and Large Neural Network Training," *CoRR*, vol. abs/2104.10350, 2021. [Online]. Available: https://arxiv.org/abs/2104.10350

[7] Meta, "Llama 2: Open Foundation and Fine-Tuned Chat Models," 2023. [Online]. Available: https://arxiv.org/abs/2307.09288

[8] I. Manotas, C. Bird, R. Zhang, D. C. Shepherd, C. Jaspan, C. Sadowski, L. L. Pollock, and J. Clause, "An Empirical Study of Practitioners' Perspectives on Green Software Engineering," in *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, L. K. Dillon, W. Visser, and L. A. Williams, Eds. ACM, 2016, pp. 237–248. [Online]. Available: https://doi.org/10.1145/2884781.2884810

[9] M. Couto, R. Pereira, F. Ribeiro, R. Rua, and J. a. Saraiva, "Towards a Green Ranking for Programming Languages," in *Proceedings of the 21st Brazilian Symposium on Programming Languages*, ser. SBLP '17. New York, NY, USA: Association for Computing Machinery, 2017. [Online]. Available: https://doi.org/10.1145/3125374.3125382

[10] R. Pereira, M. Couto, F. Ribeiro, R. Rua, J. Cunha, J. a. P. Fernandes, and J. a. 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*, ser. SLE 2017. New York, NY, USA: Association for Computing Machinery, 2017, pp. 256–267. [Online]. Available: https://doi.org/10.1145/3136014.3136031

[11] R. Pereira, M. Couto, F. Ribeiro, R. Rua, J. Cunha, J. P. Fernandes, and J. Saraiva, "Ranking Programming Languages by Energy Efficiency," *Sci. Comput. Program.*, vol. 205, p. 102609, 2021. [Online]. Available: https://doi.org/10.1016/j.scico.2021.102609

[12] S. Miller and C. Lerche. (2022) Sustainability with Rust. AWS Open Source Blog. Amazon Web Services. [Online]. Available: https://aws.amazon.com/blogs/opensource/sustainability-with-rust/

[13] ——, "Using Rust to Minimize Environmental Impact," AWS re:Invent 2021, Amazon Web Services, 2021. [Online]. Available: https://youtu.be/yQZaBtUjQ1w

[14] S. Miller, "Sustainability with Rust," AWS Summit SF 2022, Amazon Web Services, 2022. [Online]. Available: https://youtu.be/BHRLbMCpmtY

[15] W. Vogels, "AWS re:Invent 2023 Keynote," AWS re:Invent 2023, Amazon Web Services, 2023. [Online]. Available: https://youtu.be/UTRBVPvzt9w

[16] K. Mulhall and E. Collins, "Keynote: Finding Your Power to Accelerate to a Sustainable Future," KubeCon + CloudNativeCon Europe 2022, Intel, 2022. [Online]. Available: https://youtu.be/zTEN9d1Bxa8

[17] Y. Muthaiah. (2022) SAP BTP - Sustainable/Carbon Emission and energy efficiency languages. SAP Community Blog. SAP Deutschland. [Online]. Available: https://blogs.sap.com/2022/10/09/sap-btp-sustainable-carbon-emission-and-energy-efficiency-languages/

[18] G. R. D. V. Suárez, B. Bax, and A. R. Ferreres, "GreenCoding," GFT, 2021. [Online]. Available: https://www.gft.com/us/en/technology/greencoding

[19] J. Pearl, *Causality*. Cambridge, England: Cambridge University Press, 2009.

[20] I. Gouy. (2024) The Computer Language Benchmarks Game. [Online]. Available: https://benchmarksgame-team.pages.debian.net/benchmarksgame/

[21] D. Lion, A. Chiu, M. Stumm, and D. Yuan, "Investigating Managed Language Runtime Performance: Why JavaScript and Python are 8x and 29x slower than C++, yet Java and Go can be Faster?" in *2022 USENIX Annual Technical Conference, USENIX ATC 2022, Carlsbad, CA, USA, July 11-13, 2022*, J. Schindler and N. Zilberman, Eds. USENIX Association, 2022, pp. 835–852. [Online]. Available: https://www.usenix.org/conference/atc22/presentation/lion

[22] E. D. Berger, S. Stern, and J. A. Pizzorno, "Triangulating Python Performance Issues with SCALENE," in *17th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2023, Boston, MA, USA, July 10-12, 2023*, R. Geambasu and E. Nightingale, Eds. USENIX Association, 2023, pp. 51–64. [Online]. Available: https://www.usenix.org/conference/osdi23/presentation/berger

[23] L. Traini, V. Cortellessa, D. D. Pompeo, and M. Tucci, "Towards Effective Assessment of Steady State Performance in Java Software: Are We There Yet?" *Empir. Softw. Eng.*, vol. 28, no. 1, p. 13, 2023. [Online]. Available: https://doi.org/10.1007/s10664-022-10247-x

[24] E. Barrett, C. F. Bolz-Tereick, R. Killick, S. Mount, and L. Tratt, "Virtual Machine Warmup Blows Hot and Cold," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, pp. 52:1–52:27, 2017. [Online]. Available: https://doi.org/10.1145/3133876

[25] M. Hertz and E. D. Berger, "Quantifying the Performance of Garbage Collection vs. Explicit Memory Management," in *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '05. New York, NY, USA: Association for Computing Machinery, 2005, pp. 313–326. [Online]. Available: https://doi.org/10.1145/1094811.1094836

[26] Pablo Galindo Salgado. (2022) What's New In Python 3.11. Python Software Foundation. [Online]. Available: https://docs.python.org/3/whatsnew/3.11.html#faster-cpython

[27] T. Vogelsang, "Understanding the Energy Consumption of Dynamic Random Access Memories," in *43rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2010, 4-8 December 2010, Atlanta, Georgia, USA*. IEEE Computer Society, 2010, pp. 363–374. [Online]. Available: https://doi.org/10.1109/MICRO.2010.42

[28] A. Weissel and F. Bellosa, "Process Cruise Control: Event-Driven Clock Scaling for Dynamic Power Management," in *Proceedings of the 2002 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, ser. CASES '02. New York, NY, USA: Association for Computing Machinery, 2002, pp. 238–246. [Online]. Available: https://doi.org/10.1145/581630.581668

[29] M. Couto and R. Pereira, "Energy Efficiency in Programming Languages," Green Software Lab, 2017. [Online]. Available: https://github.com/greensoftwarelab/Energy-Languages

[30] K. N. Khan, M. Hirki, T. Niemi, J. K. Nurminen, and Z. Ou, "RAPL in Action: Experiences in Using RAPL for Power Measurements," *ACM Trans. Model. Perform. Evaluation Comput. Syst.*, vol. 3, no. 2, pp. 9:1–9:26, 2018. [Online]. Available: https://doi.org/10.1145/3177754

[31] Intel, *Intel 64 and IA-32 Architectures Software Developer's Manual*, Intel, 2024. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html

[32] SPEC, "SPEC CPU 2017 v1.1.9," 2022. [Online]. Available: https://www.spec.org/cpu2017/

[33] S. M. Blackburn, Z. Cai, R. Chen, X. Yang, J. Zhang, and J. Zigman, "Rethinking Java Performance Analysis," in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ser. ASPLOS '25. New York, NY, USA: Association for Computing

Machinery, 2025, pp. 940–954. [Online]. Available: https://doi.org/10.1145/3669940.3707217

[34] Stack Overflow. (2024) 2024 Stack Overflow Developer Survey. Stack Exchange. [Online]. Available: https://survey.stackoverflow.co/2024

[35] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An Updated Performance Comparison of Virtual Machines and Linux Containers," in *2015 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2015, Philadelphia, PA, USA, March 29-31, 2015*. IEEE Computer Society, 2015, pp. 171–172. [Online]. Available: https://doi.org/10.1109/ISPASS.2015.7095802

[36] H. Seol, W. Shin, J. Jang, J. Choi, J. Suh, and L. Kim, "Energy Efficient Data Encoding in DRAM Channels Exploiting Data Value Similarity," in *43rd ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2016, Seoul, South Korea, June 18-22, 2016*. IEEE Computer Society, 2016, pp. 719–730. [Online]. Available: https://doi.org/10.1109/ISCA.2016.68

[37] C. E. Leiserson, N. C. Thompson, J. S. Emer, B. C. Kuszmaul, B. W. Lampson, D. Sanchez, and T. B. Schardl, "There's Plenty of Room at the Top: What Will Drive Computer Performance After Moore's Law?" *Science*, vol. 368, no. 6495, p. eaam9744, 2020. [Online]. Available: https://www.science.org/doi/abs/10.1126/science.aam9744

[38] A. Gordillo, C. Calero, M. Á. Moraga, F. García, J. P. Fernandes, R. Abreu, and J. Saraiva, "Programming Languages Ranking Based on Energy Measurements," *Softw. Qual. J.*, vol. 32, no. 4, pp. 1539–1580, 2024. [Online]. Available: https://doi.org/10.1007/s11219-024-09690-4

[39] L. Koedijk and A. Oprescu, "Finding Significant Differences in the Energy Consumption when Comparing Programming Languages and Programs," in *International Conference on ICT for Sustainability, ICT4S 2022, Plovdiv, Bulgaria, June 13-17, 2022*. IEEE, 2022, pp. 1–12. [Online]. Available: https://doi.org/10.1109/ICT4S55073.2022.00012

[40] S. Georgiou, M. Kechagia, P. Louridas, and D. Spinellis, "What Are Your Programming Language's Energy-Delay Implications?" in *Proceedings of the 15th International Conference on Mining Software Repositories, MSR 2018, Gothenburg, Sweden, May 28-29, 2018*, A. Zaidman, Y. Kamei, and E. Hill, Eds. ACM, 2018, pp. 303–313. [Online]. Available: https://doi.org/10.1145/3196398.3196414

[41] S. Nanz and C. A. Furia, "A Comparative Study of Programming Languages in Rosetta Code," in *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, A. Bertolino, G. Canfora, and S. G. Elbaum, Eds. IEEE Computer Society, 2015, pp. 778–788. [Online]. Available: https://doi.org/10.1109/ICSE.2015.90

[42] T. Yuki and S. V. Rajopadhye, "Folklore Confirmed: Compiling for Speed = Compiling for Energy," in *Languages and Compilers for Parallel Computing - 26th International Workshop, LCPC 2013, San Jose, CA, USA, September 25-27, 2013. Revised Selected Papers*, ser. Lecture Notes in Computer Science, C. Cascaval and P. Montesinos, Eds., vol. 8664. Springer, 2013, pp. 169–184. [Online]. Available: https://doi.org/10.1007/978-3-319-09967-5_10

[43] G. Chen, K. Malkowski, M. T. Kandemir, and P. Raghavan, "Reducing Power with Performance Constraints for Parallel Sparse Applications," in *19th International Parallel and Distributed Processing Symposium (IPDPS 2005), CD-ROM / Abstracts Proceedings, 4-8 April 2005, Denver, CO, USA*. IEEE Computer Society, 2005. [Online]. Available: https://doi.org/10.1109/IPDPS.2005.378

[44] C. Hsu and W. Feng, "A Power-Aware Run-Time System for High-Performance Computing," in *Proceedings of the ACM/IEEE SC2005 Conference on High Performance Networking and Computing, November 12-18, 2005, Seattle, WA, USA, CD-Rom*. IEEE Computer Society, 2005, p. 1. [Online]. Available: https://doi.org/10.1109/SC.2005.3

[45] T. Babakol and Y. D. Liu, "Tensor-Aware Energy Accounting," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3597503.3639156

[46] S. Georgiou, M. Kechagia, T. Sharma, F. Sarro, and Y. Zou, "Green AI: Do Deep Learning Frameworks Have Different Costs?" in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22. New York, NY, USA: Association for Computing Machinery, 2022, pp. 1082–1094. [Online]. Available: https://doi.org/10.1145/3510003.3510221

[47] S. Hasan, Z. King, M. Hafiz, M. Sayagh, B. Adams, and A. Hindle, "Energy Profiles of Java Collections Classes," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: Association for Computing Machinery, 2016, pp. 225–236. [Online]. Available: https://doi.org/10.1145/2884781.2884869

[48] R. Pereira, M. Couto, J. Saraiva, J. Cunha, and J. P. Fernandes, "The Influence of the Java Collection Framework on Overall Energy Consumption," in *Proceedings of the 5th International Workshop on Green and Sustainable Software, GREENS@ICSE 2016, Austin, Texas, USA, May 16, 2016*. ACM, 2016, pp. 15–21. [Online]. Available: https://doi.org/10.1145/2896967.2896968

[49] W. Oliveira, R. Oliveira, F. Castor, G. Pinto, and J. P. Fernandes, "Improving Energy-Efficiency by Recommending Java Collections," *Empir. Softw. Eng.*, vol. 26, no. 3, p. 55, 2021. [Online]. Available: https://doi.org/10.1007/s10664-021-09950-y

[50] F. Nahrstedt, M. Karmouche, K. Bargieł, P. Banijamali, A. Nalini Pradeep Kumar, and I. Malavolta, "An Empirical Study on the Energy Usage and Performance of Pandas and Polars Data Analysis Python Libraries," in *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*, ser. EASE '24. New York, NY, USA: Association for Computing Machinery, 2024, pp. 58–68. [Online]. Available: https://doi.org/10.1145/3661167.3661203

[51] L. G. Lima, F. Soares-Neto, P. Lieuthier, F. Castor, G. Melfe, and J. P. Fernandes, "Haskell in Green Land: Analyzing the Energy Behavior of a Purely Functional Language," in *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2016, Suita, Osaka, Japan, March 14-18, 2016 - Volume 1*. IEEE Computer Society, 2016, pp. 517–528. [Online]. Available: https://doi.org/10.1109/SANER.2016.85

[52] G. Pinto, F. Castor, and Y. D. Liu, "Understanding Energy Behaviors of Thread Management Constructs," in *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, ser. OOPSLA '14. New York, NY, USA: Association for Computing Machinery, 2014, pp. 345–360. [Online]. Available: https://doi.org/10.1145/2660193.2660235

[53] B. Falsafi, R. Guerraoui, J. Picorel, and V. Trigonakis, "Unlocking Energy," in *Proceedings of the 2016 USENIX Annual Technical Conference, USENIX ATC 2016, Denver, CO, USA, June 22-24, 2016*, A. Gulati and H. Weatherspoon, Eds. USENIX Association, 2016, pp. 393–406. [Online]. Available: https://www.usenix.org/conference/atc16/technical-sessions/presentation/falsafi