Bachelor's degree in Computer Science and Engineering

Bachelor Thesis

"Evaluating performance and energy impact of programming languages"

Author
Eduardo Alarcón Navarro

Advisor
Jose Daniel García Sánchez

Leganés, Madrid, Spain
September 2025

To reach the moon, you should aim for the stars.

—

ACKNOWLEDGEMENTS

I would like to thank my family, for their continuous support and encouragement, specially this last year, where it has been the hardest for all of us. Not only they have provided me with the best education possible, financially and emotionally, but they have also taught me the importance of hard work and dedication.

I would like to thank my parents, my father for always being there when I needed him, for listening to me rant about many university projects, or crazy ideas. Furthermore, I would also like to thank my mother, for always being there for me, and for being the best mother I could have ever asked for, and for always supporting me in everything I do, even if we don't always agree on everything.

I would not have been able to finish my degree without the help of my friends, who have always been there for me, adopting me as part of their family and being there when I needed them most. During the four years of my degree, I have met many people, some have persevered in completing their degree, while others have chosen to take a different path in life. I would like to thank all of them for being there for me, and for being part of my life.

Another person I would like to thank is my thesis advisor Jose Daniel. Thank you for giving me the opportunity to work with you, after an extreme late request for a thesis. I am grateful for the trust you placed in me from the very beginning and for guiding me when I was most lost. From the admiration, it has been a great pride to close this stage with you as my advisor.

ABSTRACT

Nowadays, the importance of energy efficiency is increasing as more computationally expensive programs are being used by more and more people. The energy impact of running these programs is directly related with the programming language used to create the program as well as the design specifics.

This thesis aims to bring a specific example of the power efficiency of three programming languages: Python, Go and C++. Each one having its differences and properties, ease of use and execution speed. Each of these languages has been selected as each one has a particular characteristic that can be representative of their respective category of language.

Compiled languages with no garbage collection and no managed runtime have usually had the best execution speed as they can reach byte-code for each specific platform, but in the last years, other methods have improved significantly, such as JIT (Just in Time) compiling

To achieve realistic results, these languages were tested in multiple configurations, on different hardware, core count and operating systems to be able to eliminate any outliers.

Thus, this work will try showing the differences in energy consumption of different programming languages in a real world task, rendering a ray-traced image of multiple spheres with different materials and reflectivity.

Keywords: Compiled Language • Energy Efficiency • Interpreted Language • JIT • Ray-Tracing

CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

1.1. Motivation

Energy consumption in the software industry has been raising over the years up to a point that it is now significant at a world energy consumption.

As [1] states, in 2018 an estimated 1% of total energy consumption was attributed to datacenter alone. In 2024, it is estimated that about 1.5% of the world's energy consumption is to be blamed on data centers and server farms. These numbers may not represent much, but from the total $183,230$ TWh produced in 2023 [2] only $23,746$ TWh come from a renewable source as it can be seen from Figure 1.1, which comes to 12.96%.

Knowing which language to use for each project is decisive not only in regards to the expertise one or their team might have on that language, but also the performance and language characteristics. If you want to develop a high-performance stock trader you would never think about using a high level language such as python or Perl, but you would try sticking to compiled languages such as Java C, C++, Java or Rust.

Thus, the main motivation for this project lies in studying 3 different programming languages, with each one having peculiar characteristics, to test their respective speed and power consumption in different platforms and architectures.

This comes from the idea that the program efficiency does not come from the language itself, but the implementation of the algorithm that the programmer chooses. The language helps, but choosing the optimal algorithm is much more important.

My personal take in this project comes from my hesitation in choosing a topic to specialize in inside the Computer Science area. Having seen and used many of these languages in multiple courses along these 4 years has made me realize the importance of choosing the correct language for each problem.

2023
in terawatt-hours

Other renewables	2,428 TWh
Modern biofuels	1,318 TWh
Solar	4,264 TWh
Wind	6,040 TWh
Hydropower	11,014 TWh
Nuclear	6,824 TWh
Natural gas	40,102 TWh
Oil	54,564 TWh
Coal	45,565 TWh
Traditional biomass	11,111 TWh
Total	**183,230 TWh**

Fig. 1.1. Global electricity generation by source in 2023

1.2. Objectives

The main goal of this project is the study and analysis of three implementations of a ray-tracer program, measuring the energy consumption as well as the time each program takes to complete. It should be also noted that the platform in which the program is being run affects the energy consumption of the program.

To perform this, I have improved the code from a well-known book called Ray Tracing in One Weekend [3], translating it to go and python, updating the code so that it could handle parallel rendering.

Once the code is created, the methodology for testing the different codes need to also be created.

- x86 Intel Xeon Based
- ARM Apple Icestorm & Firestorm
- x86 Zen 2 AMD ????
- ARM Cortex-A76 CPU - Raspberry Pi 5

1.3. Document Structure

The document contains the following chapters:

- Chapter 1, *Introduction*, details the motivation of the project.

- Chapter 2, *State of the Art*, describes the main points of interest in order to fully understand the project. Theoretical and technological issues are addressed.
- Chapter 3, *Problem Statement*, general description of the project and its requirements.
- Chapter 4, *Design and Implementation*, describes the most relevant design decisions with the multi-language renderers and their multi-threaded implementation.
- Chapter 5, *Evaluation*, the analysis and benchmarks are performed and the results are exposed and discussed.
- Chapter 7, *Socioeconomic environment*, provides a comprehensive account of the project's developmental costs and its associated socio-economic implications.
- Chapter 6, *Planification*, describes the organization of the project along the development.
- Chapter 8, *Regulatory Framework*, indicates the licenses under which the project is distributed.
- Chapter 9, *Conclusions and Future Work*, briefly analyzes the results obtained and states the possible future objectives of the project.

CHAPTER 2

STATE OF THE ART

This chapter describes the paradigms and characteristics of different programming languages. Thus concepts of compiled languages, interpreters optimizations and parallelism are discussed with respect of the different programming languages.

The purpose is to provide background information necessary to understand the study and present a clear justification for the decisions made

2.1. Energy Efficient Systems

An energy efficient system is defined as a system designed and optimized to performs its functions while consuming the minimum amount of energy possible, without compromising its performance, safety and reliability.

As Muralidhar, Borovica-Gajic, and Buyya [4] put it, the average power a system draws is:

$$P_{avg} = P_{dynamic} + P_{leakage}$$

The dynamic power depends on the V supply, the clock frequency, the node capacitance and the switching activity. This power can be reduced by reducing the load on the chip or by manually setting a limit on how much voltage the chip can draw (known as undervolting [5])

2.2. System Architectures

While the energy efficiency of a system is significantly affected by connected devices (e.g., a graphics card or an AI Accelerator), this study excludes any external devices and expansion cards. Therefore, the processor architecture is the primary factor determining energy consumption on the system.

2.2.1. x86 Architecture

The x86 architecture is the most widely adopted in the world of desktop and server computers, whose market share is almost entirely shared by AMD and Intel who created it in 1978. Originally called x86-16, due to the 16 bit word size, it debuted in the Intel 8086 a single core, a $3\mu m$ node processor.

Nowadays, the technology has improved, the architecture is now called x86-64, a 64 bit extension, created by AMD, and releases the full specification in August 2000. From 2006 onward, the two companies have been developing multi-core processors, adding further Single Instruction Multiple Data (SIMD) Extensions such as AVX-512 [6]. Then came the integrated graphics and finally Power Efficiency Focus.

Then, a new paradigm came, instead of having a homogeneous set of cores, cores focused on performance and efficiency were added to the same package, the hybrid architecture. This set of heterogeneous cores meant the scheduler had to be changed in the operating systems, to better allocate more demanding programs on high performing cores and lower important tasks, such as background jobs to the highly efficient cores. This technology was released by Intel on the 12th generation Intel core processors, using Intel 7 (a $7nm$ node). This approach was revolutionary for power efficiency as Padoin, Pilla, Castro, *et al.* [7] state.

2.2.2. ARM Architecture

The ARM (Advanced RISC Machine) is the newest architecture that has reached the global scale. Developed in 1986, the goal of this new 32 bit architecture was the simplicity. As Moir [8] puts it, the energy efficient came later. This allowed the ARM architecture to dominate on the mobile sector, specially on smartphones, which run on batteries.

The characteristics of this Instruction Set Architecture (ISA) are a reduced set of instructions (Reduced Instruction Set Computer (RISC)), which allows processors to have fewer transistors than Complex Instruction Set Computer (CISC) architectures such as x86, resulting in lower cost, lower temperatures and lower power consumption.

Currently, this technology is not only used in low-power light devices, but many laptops, and even desktops are using ARM chips due to their power efficiency and performance [9].

ARM also has a hybrid technology, called big.LITTLE, as described by the [10] ARM White Paper that combines high-efficiency cores and high-performance cores. This architecture dominates the mobile device market and is increasingly found in modern laptops.

<table>
<tr><th></th><th>IVB</th><th>SKX</th><th>ZEN</th><th>ZEN2</th><th>TX2</th><th>A64FX</th></tr>
<tr><td>Manufacturer</td><td>Intel</td><td>Intel</td><td>AMD</td><td>AMD</td><td>Marvell</td><td>Fujitsu</td></tr>
<tr><td>Microarchitecture</td><td>Ivy Bridge</td><td>Skylake-X</td><td>Zen</td><td>Zen2</td><td>ThunderX2</td><td>A64FX</td></tr>
<tr><td>Instr. Set Arch.</td><td>x86 with AVX2</td><td>x86 with AVX512</td><td>x86 with AVX2</td><td>x86 with AVX2</td><td>ARMv8 with Neon</td><td>ARMv8 with SVE</td></tr>
<tr><td>Model name</td><td>Xeon E5-2690v2</td><td>Xeon Gold 6148</td><td>EPYC 7451</td><td>EPYC 7452</td><td>CN9980</td><td>PRIMEHPC FX700</td></tr>
<tr><td>Base frequency</td><td>3.0 GHz</td><td>2.4 GHz</td><td>2.3 GHz</td><td>2.35 GHz</td><td>2.5 GHz</td><td>1.8 GHz</td></tr>
<tr><td>Cores</td><td>10 per socket in one NUMA domain</td><td>20 per socket 10 per SubNUMA domain</td><td>24 per socket 6 per NUMA domain</td><td>32 per socket 8 per NUMA domain</td><td>32 per socket in one NUMA domain</td><td>48 per socket 12 per NUMA domain[1]</td></tr>
<tr><td>LD/ST reciprocal throughput</td><td>(2 LD || 1 half-ST & 1 LD || 1 half-ST) with 256 bit width per cycle</td><td>(2 LD || 1 ST & 1 LD || 1 simple-ST & 2 LD || 1 ST) with 512 bit width per cycle</td><td>(2 LD || 1 ST & 1 LD || 1 ST) with 256 bit width per cycle</td><td>(2 LD || 1 ST & 2 LD || 1 ST) with 256 bit width per cycle</td><td>(2 LD || 1 ST & 1 LD || 2 ST) with 128 bit width per cycle</td><td>(2 LD || 1 ST) with 512 bit width per cycle</td></tr>
<tr><td>LD latency</td><td>4 cy per LD</td><td>4 cy per LD</td><td>4 cy per LD</td><td>4 cy per LD</td><td>4 cy per LD</td><td>5 cy per LD</td></tr>
<tr><td>L1D cache size</td><td>32 KiB per core</td><td>32 KiB per core</td><td>32 KiB per core</td><td>32 KiB per core</td><td>32 KiB per core</td><td>64 KiB per core</td></tr>
<tr><td>L1D-L2 bandwidth</td><td>32 B/cy, half-duplex per core</td><td>64 B/cy, half-duplex per core</td><td>32 B/cy, full-duplex per core</td><td>32 B/cy, full-duplex per core</td><td>64 B/cy, half-duplex per core</td><td>64 B/cy, half-duplex per core</td></tr>
<tr><td>L2 cache size</td><td>256 KiB per core</td><td>1 MiB per core</td><td>512 KiB per core</td><td>512 KiB per core</td><td>256 KiB per core</td><td>8 MiB per core</td></tr>
<tr><td>L2-L3 bandwidth</td><td>32 B/cy, half-duplex per core</td><td>16 B/cy, full-duplex per core</td><td>32 B/cy, half-duplex per core</td><td>24 B/cy, half-duplex per core</td><td>32 B/cy, half-duplex per core</td><td>no L3</td></tr>
<tr><td>L3 cache type</td><td>inclusive</td><td>victim</td><td>inclusive</td><td>victim</td><td>victim</td><td>no L3</td></tr>
<tr><td>L3 cache size</td><td>25 MiB per per socket</td><td>13.75 MiB per SubNUMA domain</td><td>8 MiB per CCX with two CCXs per NUMA domain</td><td>16 MiB per NUMA domain</td><td>32 MiB per socket</td><td>no L3</td></tr>
<tr><td>Memory bandwidth</td><td>56 GB/s with load per full socket</td><td>131 GB/s with load per full socket</td><td>149 GB/s with load per full socket</td><td>143 GB/s with load per full socket</td><td>125 GB/s with load per full socket</td><td>227 GB/s with load per full socket</td></tr>
</table>

[1] modeled in KERNCRAFT as 4 sockets with 12 cores each

Fig. 2.1. Architectures and most important parameters used for evaluation

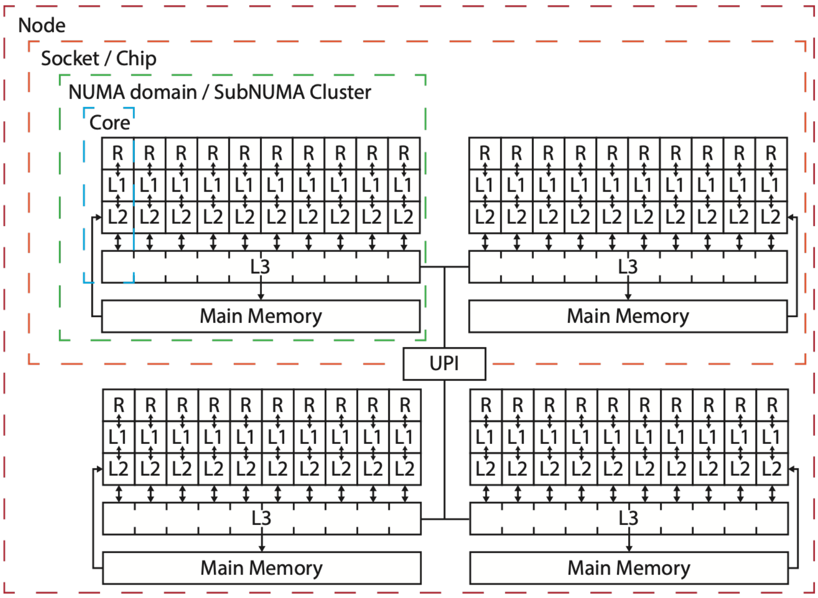

Fig. 2.2. Intel Xeon Skylake-X

2.2.3. CPU Design for Multi-core dies

Is it important to note that a computer that reports having more than 32 logical cores, usually has more than one socket, thus the performance and scaling of programs on multiple sockets can affect the energy efficiency and performance. This is due to the fact that information has to move between the multiple cache levels.

From [11]'s Figure 2.1 we can see there are multiple configurations, depending on the architecture, the amount of cache per core, how many cores there are per chip and the memory bandwidth.

The traditional layout of these Central Processing Units (CPUs) can be found from Figure 2.2, where each ten cores form a Non-Uniform Memory Access (NUMA) domain, two NUMA domains for each of the chips (sockets) and multiple chips (two in this case) form a NUMA node.

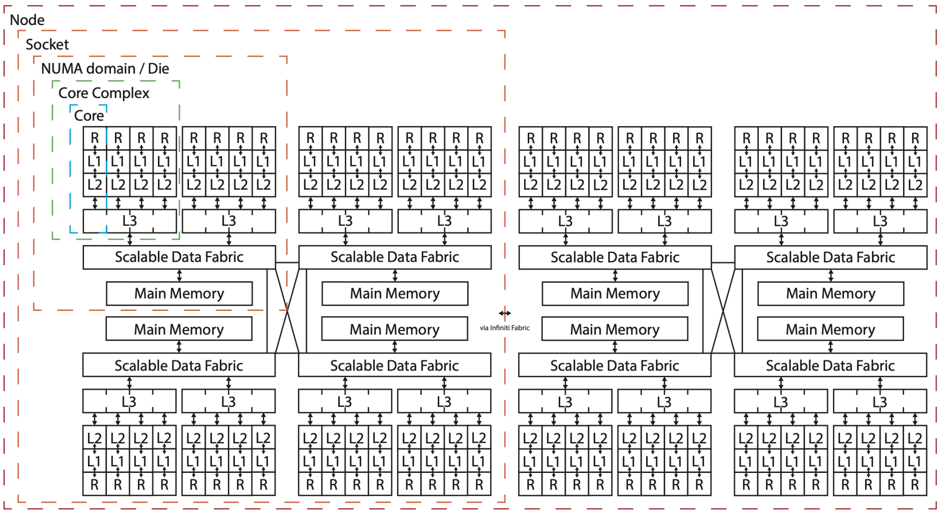

Fig. 2.3. AMD introduced an additional hierarchy level with its Zen architecture: CCX. A single core complex has up to four cores, with a sliced victim L3 cache. Two CCXs are combined onto a die, sharing a local partition of main memory. Multiple dies make up a socket

Using AMD's ZEN 2 architecture as a new CPU architecture example, we can observe there are is an additional layer, compared to Figure 2.2. As Gibbs [11] puts it in their paper, in 2.3, the victim of this design is the L3 cache, which is shared between less cores.

2.3. Programming Languages

In this section, the different programming languages that have been chosen will be discussed, as well as why other similar languages were not.

2.3.1. Go: Compiled Language with an Embedded Managed Runtime

Go is a language developed by Google, released in 2009, focused on concurrency. It has a runtime which manages the goroutines. As described by [12], Go has a Garbage Collector, which means while the program is running, there needs to be a thread checking for unused memory structures.

This language was designed for backend tasks, handling thousands of simultaneous connections, while having an easy syntax for any programmer. Some companies that use this are Uber, Docker, Twitch, previously Discord [13] and, although not mainly, Netflix.

From Table 2.1, it can be seen that Go is statically typed and compiled, which makes it have a good start as an efficient programming language. But from the Table 2.2, we can observe go has a managed runtime, which means the energy consumption will be

TABLE 2.1
COMPARISON OF C++, PYTHON, AND GO GENERAL CHARACTERISTICS

Characteristic	C++	Python	Go
Typing System	Statically Typed: Types are checked at compile-time, catching errors early and aiding optimization.	Dynamically Typed: Type checking occurs at runtime. Optional static type hinting (PEP 484) is available.	Statically Typed: Types are checked at compile-time, ensuring type safety and early error detection.
Compilation & Execution	Compiled: Code is compiled directly to native machine code for fast execution.	Interpreted: Typically compiled to bytecode which is then executed by a VM.	Compiled: Code is compiled directly to a self-contained native machine code executable with a runtime.
Concurrency	Provides low-level primitives like threads and mutexes, requiring manual management.	Offers threading (limited by the GIL for CPU-bound tasks) and multiprocessing libraries.	Built-in support with lightweight goroutines and channels managed by the Go runtime.
Memory Management	Manual memory management, with modern C++ heavily relying on RAII and smart pointers.	Automatic via reference counting and a cyclic garbage collector.	Automatic via a concurrent, tri-color mark and sweep garbage collector.
Standard Library	Rich library with a focus on performance (e.g., STL containers, algorithms).	Extensive "batteries-included" library for a vast array of tasks, speeding up development.	Comprehensive library designed for modern needs like networking, I/O, and JSON handling.
Programming Paradigms	Multi-paradigm: Supports procedural, object-oriented (OOP), and generic programming.	Multi-paradigm: Supports procedural, object-oriented, and functional styles.	Primarily procedural and concurrent. Uses composition over inheritance (no classes).

TABLE 2.2
COMPARISON OF LANGUAGE CHARACTERISTICS IMPACTING ENERGY EFFICIENCY

Characteristics Impacting Performance & Energy Efficiency			
Characteristic	**C++**	**Python**	**Go**
Typing System	Static typing and templates enable compile-time code specialization, avoiding runtime polymorphism overhead.	Dynamic typing limits static optimizations, as type checks and memory allocation occur at runtime.	Static typing allows compiler optimizations like devirtualization and function inlining.
Execution & Compilation	Mature compilers generate highly optimized machine code, leading to shorter active CPU time and lower energy use.	Code is compiled to bytecode and run on a VM. This interpreter overhead significantly impacts performance.	Compiles to efficient native machine code but needs a runtime. The compiler performs optimizations for performance.
Concurrency Model	Low-level primitives (`std::thread`) offer fine-grained control without a GIL but require manual management.	Threading is limited by the GIL for CPU-bound tasks. Multiprocessing works but has higher overhead.	Lightweight goroutines and channels allow for high concurrency with very low overhead, managed by the runtime.
Memory Management	Manual memory control (`new`/`delete`, smart pointers) and RAII provide deterministic cleanup, avoiding GC overhead.	Automatic GC on mostly heap-allocated objects increases memory footprint, GC load, and access latency.	Automatic GC with a focus on stack allocation for value types, which reduces GC pressure and improves data locality.
Standard Library	The Standard Template Library (STL) provides highly optimized, performance-focused data structures and algorithms.	Performance-critical modules are often C extensions, but the call overhead from Python remains.	Many standard library functions (e.g., networking, crypto) are highly optimized, some using assembly for critical paths.
Abstractions	Aims for "zero-cost abstractions," where high-level features are compiled away and incur no runtime overhead.	High-level abstractions and dynamic features are powerful but generally incur significant runtime overhead.	Interfaces provide abstraction with a small, well-defined runtime cost. Composition is favored over inheritance.

higher than other languages that do not have this. This runtime is the section of the program in charge of running and scheduling goroutines. This is why go binaries have a bigger minimum size as the runtime has to be fitted in the binary, which is great for cross compilation, but not great for either energy efficiency or performance.

Go's scheduler performs a series of steps before starting to run the user's code. As described in [14], [15] and [16], the runtime can be divided into these steps:

1. **OS Loading**: The main function is not the actual entry point of a Go program. Rather, the starting point is an assembly level function within the runtime. You can find it in a file corresponding to your specific OS and architecture, for example, `rt0_linux_amd64.s`. This is the first function the Go's program code will have the OS execute after loading the binary, and its only responsibility is to get the environment setup for the Go runtime.

2. **Argument and Environment setup**: The runtime, after being loaded into memory, calls an internal function `runtime.args` that handles the arguments and environment. This function copies the arguments (`argc` and `argv`) and environment variables into a Go-managed memory space. This ensures that the rest of the Go program, including the main function, can access this information through standard library functions like `os.Args` and `os.Getenv`.

3. **Scheduler Initialization (M:P:G Model)**: The heart of the concurrency system in Go is the "M:G:P" model. Before any go code is executed, the scheduler must be initialized, which happens inside `runtime.schedinit`.

   - **`M0` and `G0` Creation**: The program starts with a single Operating System (OS) thread ($M0$). Every M thread has a special goroutine called `g0`, which is responsible for scheduling other runtime tasks.
   - **P Initializations**: A list of `Ps` or processors, which is a resource required to execute Go code, is created. The limit of `P` is determined by the `GOMAXPROCS` environment variable or inside the Go code by using the `runtime.GOMAXPROCS()` function.

   At this time, the scheduler limits are put in place, limiting the maximum number of OS threads to 10,000.

4. **Memory Allocator and GC initialization**: Go's runtime includes a complex memory allocator and Garbage Collector [17] and [18] .

   - **Memory Reservation**: The runtime reserves a large region of virtual memory, divided into 3 areas: `spans`, `bitmap` and `arena` (where go objects are allocated on the heap.
   - **Allocator Structures**: Other structures such as the `mheap` (the global heap structure for Go, `mcentral` (a central cache for memory spans) and for each

P a per-thread cache for allocating small objects without locking the main thread, called `mcache`.

- **GC Pacer**: The pacer determines the optimal time to trigger a Garbage Collection cycle based on the `GOGC` environment variable. The goal of Go's collection system is to perform one as the heap doubles in size since the previous cycle.

5. **Package Initialization**: At this point, the runtime can start reading from the supplied files. It starts with importing the required dependencies and initializing package-level variables. Once all files are processed in lexical file name order, the `init()` function or functions are called in order.

6. **Creating the Main Goroutine**: The runtime doesn't call the `main.main` function directly. Instead, it creates a new goroutine to execute it. This is done using the internal `runtime.newproc` function. A new goroutine (G) is created, and its instruction pointer is set to the `main.main` function. This new goroutine is then placed into the local run queue of one of the available Ps, making it runnable.

7. **Start the Scheduler**: Finally, the `runtime.mstart` is called on the main thread, that enters into the scheduling loop. From this point on, the Go program is running, and the scheduler is fully operational, managing the execution of all goroutines on the available threads.

2.3.2. Python: Interpreted Programming Language

Python is the most popular language according to the TIOBE Index as of May 2025 and has been since October 2021. Either because of its easy to start as a simple to start with programming language or because the actual trend of Artificial Intelligence (AI) is mostly programmed with Python, its popularity has skyrocketed.

Python, on the contrary to most of the other popular languages is an interpreted language, which means instead of having a compiler turn the code into assembly and then into binary, it has an interpreter that runs the bytecode instructions written by the programmer one by one.

As [19] puts it: "Garbage collection also can have a significant impact on both execution time and memory usage, and can be fine-tuned to obtain better performance"

"Compiling" Python code to bytecode

There are multiple steps between the Python code that the user writes and its execution, which can be divided into two phases [20] on CPython, the default python interpreter:

1. **Phase 1: The "Frontend"**

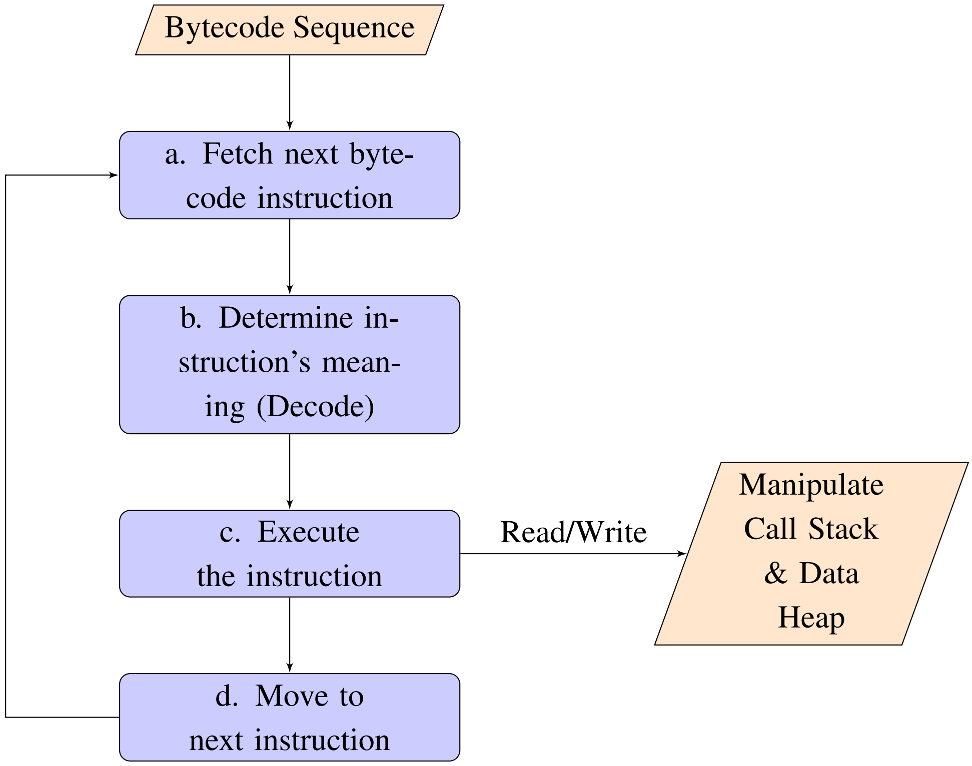

Fig. 2.4. Python's Execution loop

(a) **Reading the source code**: The python interpreter reads the *.py* files that were passed as an argument when invoked.

(b) **Lexical Analysis (Lexing & Tokenizing)**: In this step, the interpreter breaks down the code into a sequence of tokens, the smallest meaningful unit of the language's grammar.

(c) **Parsing**: The stream of tokens from the previous step is fed into the parser, which checks if the sequence of tokens conforms to the rules that define Python's grammar [21].

(d) **Compiling to bytecode**: The Abstract Syntax Tree (AST) is traversed generating bytecode. When the compilation is done, python caches the bytecode into a folder names `__pycache__` and stores `.pyc` files, representing the bytecode version of the same named file, so that in future executions, all the previous steps can be skipped.

2. **Phase 2: The "Backend"**

(a) **Loading the bytecode** into the Python's Virtual Machine (PVM), either from the output of the compiler or from the `.pyc` file.

(b) **Python's Execution loop**: As we can see form Figure 2.4, the loop is extremely simple. It starts by fetching the next instruction, decoding that instruction, executing it and moving the pointer to the next instruction.

(c) **Stack Frame Management**: The PVM manages function execution by pushing a stack frame, containing the function's context like its variables and return address, onto the call stack upon invocation and popping it upon completion to resume the prior state.

(d) **Memory Management**:

- Reference Counting: This is the primary mechanism. It works by having all objects keep a count of how many variables or other objects refer to themselves. If this count drops to zero, the object is removed from memory and thus that section of the memory is freed.
- Cyclic Garbage Collector: As there are some cases where the reference counting can not deal with cyclic references (e.g., when object α refers to β and β refers to α), a garbage collector process also has to run periodically. This means the efficiency of the interpreter is not very high as it need extra processes to clean up memory. This GC uses a generational approach, based on the idea that most objects are short-lived, and focuses its effort on newer objects. [1]

A Concrete Example with `dis`

Listing 2.1. Python code demonstrating the `dis` module.

```python
import dis

def simple_math(a):
    x = a + 10
    return x

# Use the disassembler to inspect the function's bytecode
dis.dis(simple_math)
```

Let's see this in action. The `dis` module is a "disassembler" that shows you the bytecode for a piece of Python code. The script in Listing 2.1 defines a simple function and then uses `dis` to inspect it.

Listing 2.2. Bytecode output generated by the `dis.dis` function.

```
  4           0 LOAD_FAST                0 (a)
              2 LOAD_CONST               1 (10)
              4 BINARY_ADD
              6 STORE_FAST               1 (x)

  5           8 LOAD_FAST                1 (x)
```

[1] In some interpreters such as CPython, you can interact with the collector using the **gc** module. [22]

```
                10 RETURN_VALUE
```

The output of this script, shown in Listing 2.2, reveals the low-level bytecode instructions that the Python Virtual Machine will execute.

Other Interpreters

TABLE 2.3
ALTERNATIVE PYTHON IMPLEMENTATIONS

Implementation	Description
IronPython	Python running on .NET
MicroPython	Python running on microcontrollers and in the Web browser
Stackless Python	A branch of CPython supporting microthreads
Jython	Python running on the Java Virtual Machine

As python's interpreter is almost completely independent from it's syntax and language development, there are multiple interpreter, each one with its features. One of the most popular alternatives to CPython is PyPy, a fast implementation of Python with a Just In Time Compiler (JIT) compiler. The problem with Just in Time compilers are that there might incurr into potential warmup costs, before the functions go though the compiler. This process can optimize some hot code paths (a function or a section of the code that is run multiple times). Other examples are shown out in Table 2.3

2.3.3. C++: Directly Compiled, Unmanaged Language

C++ is one of the most famous language when it comes to high performance computing applications. Based on the programming language C, released in 1978 as a high-level language at the time, compared to assembly.

As we can see from Table 2.1, there are many characteristics on why the language is one of the most used for high performance software, for example, blender or nuke. This known examples and the multiple tests performed in multiple courses during the computer science degree.

If we take into account the energy efficiency, from Table 2.2 we can observe that being a compiled language, with multiple optimizations at the compilation level, zero-cost abstractions, no runtime and direct memory management makes it one of the best low energy consumption language in theory. In this section, the different programming languages that have been chosen will be discussed, as well as why other similar languages were not.

Compilers

There are two main compilers for C++ widely used in the industry Clang++ and G++:

G++ is the C++ compiler for the GNU Compiler Collection (GCC). It is widely considered as a seasoned, reliable veteran; it's the default on most Linux distributions and has a long history of producing highly optimized code for final release builds. While its error messages have improved significantly over the years, they can sometimes be verbose and a bit cryptic, leaving you to decipher a long template expansion error.

Clang++ is the C++ compiler front-end for the LLVM project. It often feels like it was designed specifically to make a developer's life easier, excelling in two key areas: speed and diagnostics. Clang++ is famous for its remarkably fast compilation times and for error messages that are not only clear and color-coded but often suggest the exact fix, creating a much tighter and less frustrating coding loop. This focus on tooling is why it's also the engine behind many modern IDE features and static analysis tools.

2.3.4. Other languages not used

There are many more languages, but to reduce the scope of the project and have a good analysis on each of the languages to be analyzed, a reduced group had to be selected.

As a contender for a fast, high energy efficient language we could have used Rust, a recently new programming language, focused in performance and type-safety. As Rust is a compiled language and uses the same LLVM backend for compilation, a similar result is to be expected from this benchmark compared to the C++ implementation.

Other programming languages that could have been good contenders to be tested, not because of their efficiency but because of their widespread use could have been:

2.4. Previous Benchmarks

Research in this area has intensified recently, driven by the growing global imperative to improve energy efficiency. This topic is no longer a niche concern but has garnered significant interest across industrial, economic, and policy-making sectors worldwide.

One of the first papers published that sparked the flame in the energy efficiency aspect of software development is Pereira, Couto, Ribeiro, *et al.* [23], that analyzed the performance of twenty seven languages in ten different programming problems. The problem I found out with these kinds of tests is that the algorithms usually are quite simple and do not represent real-world applications.

As Kempen, Kwon, Nguyen, *et al.* [24] put it, "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 di-

TABLE 2.4
LANGUAGES EXCLUDED FROM THE STUDY AND JUSTIFICATION

Language	Reason for Exclusion
Java / C#	These languages primarily execute on managed runtimes (the JVM and .NET CLR, respectively). Their common Just-In-Time (JIT) compilation model is fundamentally different from the Ahead-Of-Time (AOT) native compilation of C++ and Go. Including them would be similar to go's implementation with a specific runtime .
JavaScript / TypeScript	These language was designed for more web-centric environments, these languages run on JavaScript engines and typically use a single-threaded event loop for concurrency. This distinct execution paradigm and primary application domain fall outside the scope of this study, which focuses on general-purpose compiled languages. Some runtimes that Javascipt use are Node, Deno or bun, but all of them have to use a core, either V8 (for node and Deno) or JavaScriptCode (for bun).
Ada	While a statically typed and being Ahead-of-time compiled language, Ada is highly specialized for high-integrity, real-time, and safety-critical systems (e.g., avionics, defense). Its lower mainstream adoption and niche focus make it less representative for this study.
Zig	As a modern systems language, Zig aligns well with the technical characteristics of C++ and Go. However, it is a relatively new language that has not yet achieved the same level of industrial adoption, ecosystem maturity, or long-term stability as the selected languages. The study's focus is on established, widely-used technologies to ensure the relevance of the findings to the current software development landscape.

rect 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 [25]". Continuing with that idea, they also state that short benchmarks are not realistic of a real-life program being executed on a machine.

The most important aspect of comparing two languages when doing a benchmark, is making sure that the algorithm used to solve the task is the same between the multiple languages. Thus, the benchmarks that test the languages using libraries sometimes poison the results by adding another language without their knowledge. For instance, if the python program uses `numpy`, most of that library is implemented in C, or if we were to use `PyTorch`, that library is mostly implemented in C++-

Other studies such as [25], specifically on their Table 3, on the top languages, for `binary-trees`, `fannkuch-redux`, and `fasta`, are C, C++, Rust and fortran, exchanging positions depending on the test. As we see from this list, these are all compiled languages, which is not a surprise, as Abdulsalam, Lakomski, Gu, *et al.* [26] comment on their paper comparing multiple compiler flags and other languages.

As Lion, Chiu, Stumm, *et al.* [27] state in their publication "studies have found that V8 and CPython can be 8.01x and 29.50x slower on average that their C++ counterparts respectively" but they also state that, "choosing a language for your application simple because it is 'fast' is the ultimate form of premature optimization" [28]. They have also found out that with respect to a language with a runtime, it can provide a better performance and scalability. Although this might be counter-intuitive, Go's scheduler automatically maps user threads to kernel threads thus reducing the number of context switches.

Another challenge this area faces is measuring the energy consumption. Some CPUs have internal registers that can provide these measurements, like the intel RAPL, but other machines might have those registers not accessible to the user or, in case of most ARM processors, these registers do not exist. Another technique is using the measurement from an external power plug, like the TP-link HS110, used by Søndergaard, Nielsen, Jensen, *et al.* [29] in their measurements. The problem with this kinds of measurement devices it that the entire system is measured instead of only the CPU or the Random Acces Memory (RAM) and the precision it provides is much lower, being able to read only at a 1 Hz frequency.

CHAPTER 3

PROBLEM STATEMENT

3.1. Project Description

3.2. Requirements

3.2.1. Functional Requirements

3.2.2. Non Functional Requirements

3.3. Use Case

3.4. Traceability

CHAPTER 4

DESIGN AND IMPLEMENTATION

4.1. General Program Design

4.2. Scene

4.2.1. Sphere_data design

4.2.2. Language Specific

C++

Go

Python

4.3. Object

C++

Go

Python

4.4. Renderer

4.4.1. Camera

C++

Go

Python

4.4.2. Multiprocessing

C++

Go

Python

4.5. Output

CHAPTER 5

EVALUATION

5.1. Measurement Platforms

5.1.1. Many Core Platform

Evaluated parameters

Results

5.1.2. Personal Desktop

Evaluated parameters

Results

5.1.3. Personal SOTA Laptop

Evaluated parameters

Results

5.1.4. Raspberry Pi 5

Evaluated parameters

Results

CHAPTER 6

PLANIFICATION

CHAPTER 7

SOCIOECONOMIC ENVIRONMENT

7.1. Budget

7.1.1. Human Resources

7.1.2. Material Resources

Hardware

7.1.3. Software

7.1.4. Indirect Costs

7.1.5. Total Cost

7.2. Socio-Economic Impact

CHAPTER 8

REGULATORY FRAMEWORK

CHAPTER 9

CONCLUSIONS AND FUTURE WORK

Example of figure:

Fig. 9.1. Figure name here

Example of table:

TABLE 9.1

LOREM IPSUM

I		II		III			IV
x	y	x	y	x	y	x	y
10.0	8.04	10.0	9.14	10.0	7.46	8.0	6.58
8.0	6.95	8.0	8.14	8.0	6.77	8.0	5.76
13.0	7.58	13.0	8.74	13.0	12.74	8.0	7.71
9.0	8.81	9.0	8.77	9.0	7.11	8.0	8.84
11.0	8.33	11.0	9.26	11.0	7.81	8.0	8.47
14.0	9.96	14.0	8.10	14.0	8.84	8.0	7.04
6.0	7.24	6.0	6.13	6.0	6.08	8.0	5.25
4.0	4.26	4.0	3.10	4.0	5.39	19.0	12.50
12.0	10.84	12.0	9.13	12.0	8.15	8.0	5.56
7.0	4.82	7.0	7.26	7.0	6.42	8.0	7.91
5.0	5.68	5.0	4.74	5.0	5.73	8.0	6.89

Source: BOE

BIBLIOGRAPHY

[1] 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. DOI: 10.1126/science.aba3758. eprint: https://www.science.org/doi/pdf/10.1126/science.aba3758. [Online]. Available: https://www.science.org/doi/abs/10.1126/science.aba3758.

[2] H. Ritchie, P. Rosado, and M. Roser. "Energy production and consumption." (Jan. 2024), [Online]. Available: https://ourworldindata.org/energy-production-consumption.

[3] P. Shirley, T. D. Black, and S. Hollasch. "Ray tracing in one weekend." (Apr. 2025), [Online]. Available: https://raytracing.github.io/books/RayTracingInOneWeekend.html.

[4] R. Muralidhar, R. Borovica-Gajic, and R. Buyya, "Energy efficient computing systems: Architectures, abstractions and modeling to techniques and standards," *CoRR*, vol. abs/2007.09976, 2020. arXiv: 2007.09976. [Online]. Available: https://arxiv.org/abs/2007.09976.

[5] A. Muc, T. Muchowski, M. Kluczyk2, and A. Szeleziński, "Analysis of the use of undervolting to reduce electricity consumption and environmental impact of computers," *Rocznik Ochrona Środowiska*, 2020.

[6] Intel. "Maximize roi and performance for demanding workloads with intel avx-512." (Jun. 2024), [Online]. Available: https://www.intel.com/content/www/us/en/products/docs/accelerator-engines/what-is-intel-avx-512.html.

[7] E. Padoin, L. L. Pilla, M. Castro, F. Z. Boito, P. Navaux, and J.-F. Méhaut, "Performance/energy trade-off in scientific computing: The case of arm big.little and intel sandy bridge," *IET Computers & Digital Techniques*, vol. 9, pp. 27–35, Dec. 2014. DOI: 10.1049/iet-cdt.2014.0074.

[8] C. Moir. "The remarkable story of arm." (Jun. 2020), [Online]. Available: https://medium.com/swlh/the-remarkable-story-of-arm-85760399c38d.

[9] W. Wolff and B. Porter, "Performance optimization on big.little architectures: A memory-latency aware approach," in *The 21st ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, ser. LCTES '20, London, United Kingdom: Association for Computing Machinery, 2020, pp. 51–

61. DOI: 10.1145/3372799.3394370. [Online]. Available: https://doi.org/10.1145/3372799.3394370.

[10] A. Holdings, "Big.little technology: The future of mobile," ARM Limited, White Paper, Sep. 2013. [Online]. Available: https://armkeil.blob.core.windows.net/developer/Files/pdf/white-paper/big-little-technology-the-future-of-mobile.pdf (visited on 06/07/2025).

[11] M. Gibbs, "Design and implementation of pay for performance," *SSRN Electronic Journal*, pp. 35–36, Feb. 2012. DOI: 10.2139/ssrn.2003655.

[12] M. Rosecrance. "The garbage collector." (Feb. 11, 2019), [Online]. Available: https://www.youtube.com/watch?v=gPxFOMuhnUU (visited on 06/05/2025).

[13] J. Howarth. "Why discord is switching from go to rust." (2020), [Online]. Available: https://discord.com/blog/why-discord-is-switching-from-go-to-rust (visited on 03/15/2025).

[14] J. Espino, *Understanding the go runtime*, 2023. [Online]. Available: https://golab.io/talks/understanding-the-go-runtime (visited on 06/07/2025).

[15] W. Kennedy. "Scheduling in go : Part ii - go scheduler." (2025), [Online]. Available: https://www.ardanlabs.com/blog/2018/08/scheduling-in-go-part2.html (visited on 06/07/2025).

[16] T. Liu. "Memory allocations." (2020), [Online]. Available: https://go101.org/optimizations/0.3-memory-allocations.html (visited on 06/07/2025).

[17] SoByte. "Principle of memory allocator implementation in go language." (2021), [Online]. Available: https://www.sobyte.net/post/2021-12/golang-memory-allocator/ (visited on 06/08/2025).

[18] S. Matsiukevich. "Golang internals, part 6: Bootstrapping and memory allocator initialization." (2015), [Online]. Available: https://www.altoros.com/blog/golang-internals-part-6-bootstrapping-and-memory-allocator-initialization/ (visited on 06/08/2025).

[19] M. Hertz and E. D. Berger, "Quantifying the performance of garbage collection vs. explicit memory management," *SIGPLAN Not.*, vol. 40, no. 10, pp. 313–326, Oct. 2005. DOI: 10.1145/1103845.1094836. [Online]. Available: https://doi.org/10.1145/1103845.1094836.

[20] P. S. Foundation. "Cpython internal documentation." (2025), [Online]. Available: https://github.com/python/cpython/tree/main/InternalDocs (visited on 06/07/2025).

[21] P. S. Foundation. "Full grammar specification." Part of the Python Language Reference for Python 3. (2025), [Online]. Available: https://docs.python.org/3/reference/grammar.html (visited on 06/07/2025).

[22] P. S. Foundation. "Gc - garbage collector interface." (2025), [Online]. Available: https://docs.python.org/3/library/gc.html (visited on 06/07/2025).

[23] R. Pereira *et al.*, "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, Vancouver, BC, Canada: Association for Computing Machinery, 2017, pp. 256–267. DOI: 10.1145/3136014.3136031. [Online]. Available: https://doi.org/10.1145/3136014.3136031.

[24] 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*, 2025. DOI: 10.48550/arXiv.2410.05460. arXiv: 2410.05460 [cs.PL]. [Online]. Available: https://arxiv.org/abs/2410.05460.

[25] R. Pereira *et al.*, "Ranking programming languages by energy efficiency," *Science of Computer Programming*, vol. 205, p. 102 609, 2021. DOI: https://doi.org/10.1016/j.scico.2021.102609. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167642321000022.

[26] S. Abdulsalam, D. Lakomski, Q. Gu, T. Jin, and Z. Zong, "Program energy efficiency: The impact of language, compiler and implementation choices," in *International Green Computing Conference*, 2014, pp. 1–6. DOI: 10.1109/IGCC.2014.7039169.

[27] 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 22)*, Carlsbad, CA: USENIX Association, Jul. 2022, pp. 835–852. [Online]. Available: https://www.usenix.org/conference/atc22/presentation/lion.

[28] N. Humrich. "Yes, python is slow, and i don't care." (2017), [Online]. Available: https://medium.com/pyslackers/yes-python-is-slow-and-i-dont-care-13763980b5a1 (visited on 03/15/2025).

[29] J. Z. Søndergaard, M. C. B. Nielsen, S. A. B. Jensen, and V. F. J. and, "Measuring energy efficiency of jit compiled programming languages with micro-benchmarks," Aalborg University, Tech. Rep., 2025.

GLOSSARY

A

AI Accelerator

An expansion card that usually has specific hardware to accelerate AI workloads. 5

AST

An abstract syntax tree is a data structure used in computer science to represent the structure of a program or code snippet. It is a tree representation of the abstract syntactic structure of text (often source code) written in a formal language.. 13

B

bytecode

Bytecode is a set of instructions for a hypothetical machine, for example the Python Virtual Machine (PVM). or the Java Virtual Machine (JVM). 9, 10, 12, 13

C

C extension

TODO. 10

channel

TODO. 9

CPython

Pythons default interpreter, with a GIL, until version 3.13 which an experimental free-threaded version was created and version 3.14 will continue its development (both known as 3.13t and 3.14t respectively). 12, 14, 15, 18

cross compilation

Compiling a program with a different architecture than the host machine that is making the compilation.. 11

F

fortran

Fortran was a pioneering, high-level compiled programming language (short for Formula Translation) developed in the 1950s, which remains widely used for its high performance in scientific, engineering, and numerical computation.. 18

G

Garbage Collection cycle

A Garbage Collection Cycle is a single execution of the garbage collection system that identifies and reclaims memory from objects no longer in use by an application.. 12

goroutine

A goroutine is a lightweight thread managed by the Go runtime. 8, 9, 11, 12

L

LLVM

LLVM began as a research project at the University of Illinois, with the goal of providing a modern compilation strategy capable of supporting both static and dynamic compilation of arbitrary programming languages. The LLVM Project is a collection of modular and reusable compiler and toolchain technologies. Despite its name, LLVM has little to do with traditional virtual machines. The name "LLVM" itself is not an acronym; it is the full name of the project.. 16

M

mutex

TODO. 9

N

NUMA

It's a computer memory design where the time it takes to access memory depends on which processor is accessing which memory location. 7

numpy

The most popular python package for scientific computation with python. 18

P

PyTorch

An open-source machine learning framework for Python, providing GPU-accelerated tensor computation and automatic differentiation for building and training deep neural networks.. 18

R

RISC

A Reduced Instruction Set Computer is a type of microprocessor architecture that utilizes a small, highly-optimized set of instructions rather than the highly-specialized set of instructions typically found in other architectures.. 6

T

tri-color mark and sweep

TODO. 9

V

V8

Google's open-source, high-performance JavaScript and WebAssembly engine, written in C++, that powers the Google Chrome browser and the Node.js runtime environment. 17, 18

ACRONYMS

A

AI

Artificial Intelligence. 12

AST

Abstract Syntax Tree. 13, *Glossary:* AST

C

CCX

Core Complex. xi, 8

CISC

Complex Instruction Set Computer. 6

CPU

Central Processing Unit. 7, 9, 10, 18

G

GC

Garbage Collector. 10–12, 14, *Glossary:* GC

GIL

Global Interpreter Lock. 9, 10, 37, *Glossary:* GIL

I

I/O

Input / Output. 9

ISA

Instruction Set Architecture. 6

J

JIT

Just In Time Compiler. 15

JSON

JavaScript Object Notation. 9

N

NUMA

Non-Uniform Memory Access. 7, *Glossary:* NUMA

O

OOP

Object-Oriented Programming. 9, *Glossary:* OOP

OS

Operating System. 11

P

PVM

Python's Virtual Machine. 13, 14

R

RAII

Resource Acquisition Is Initialization. 9, 10

RAM

Random Acces Memory. 18

RISC

Reduced Instruction Set Computer. 6, *Glossary:* RISC

S

SIMD

Single Instruction Multiple Data. 6

V

VM

Virtual Machine. 9, 10, *Glossary:* VM