Investigating Managed Language Runtime Performance: Why JavaScript and Python are 8x and 29x slower than C++, yet Java and Go can be Faster?

David Lion, *University of Toronto and YScope Inc.;* Adrian Chiu and Michael Stumm, *University of Toronto;* Ding Yuan, *University of Toronto and YScope Inc.*

https://www.usenix.org/conference/atc22/presentation/lion

This paper is included in the Proceedings of the 2022 USENIX Annual Technical Conference.

July 11–13, 2022 • Carlsbad, CA, USA

978-1-939133-29-8

Investigating Managed Language Runtime Performance

Why JavaScript and Python are 8x and 29x slower than C++, yet Java and Go can be faster?

David Lion*†, Adrian Chiu*, Michael Stumm*, Ding Yuan*†
*University of Toronto, †YScope

Abstract

The most widely used programming languages today are managed languages. They are popular because their vast features improve many aspects of code development, including increased productivity and safety. However, as a product or service scales in usage, performance issues become a problem. Developers are then often faced with complex choices as they must decide whether the desired performance can be squeezed from existing code, or whether their language has reached its performance limits, requiring years of code to be ported to a new more-performant language. To make matters worse, runtime performance is shrouded in mystery as it involves complex interactions of different components, such as interpreter, just-in-time (JIT) compiler, thread library, and Garbage Collection (GC) system.

We present an in-depth performance analysis and comparison of Java, Go, JavaScript, and Python, using C++ as a baseline. We carefully instrumented the different language runtimes, so that developers can precisely measure the number of cycles taken to execute any bytecode instruction, or the overhead of dynamic type checking in JavaScript. This allows us to accurately identify sources of overhead. We further created 6 applications from the ground up to establish the *LangBench* benchmark; the applications cover a range of complexity, and they cover a variety of application scenarios differing in compute intensity, memory usage, network and disk I/O intensity, and available concurrency. We comprehensively analyze their completion times, resource usage, and scalability.

Overall, we found that V8/Node.js and CPython exhibit excessive overheads, executing applications 8.01x and 29.50x slower on average than their C++ counterparts, respectively. Making matters worse, applications on these two runtimes scale poorly in that they cannot effectively utilize more than one core. In contrast, OpenJDK and Go applications are performance competitive to C++, running only 1.43x and 1.30x slower, respectively, and they can easily scale to multiple cores. There are applications where OpenJDK and Go outperform their C++ counterparts.

1 Introduction

Programming languages with integrated run-time environments have continuously grown in popularity. The three most popular languages on GitHub since 2015 are JavaScript, Java, and Python [38]. These languages offer the promise of improved developer productivity and thus faster product creation and adaptation because of a variety of features they offer, including easier readability and usability, dynamic type checking, memory management with garbage collection, and dynamic memory safety checks. We use the term "*managed languages*" to refer to these type of programming languages.

Managed languages are increasingly being used to implement *systems software* where performance is critical. Both Hadoop [54] and Spark [76] run on a Java Virtual Machine (JVM) [61] as they are implemented in Java and Scala respectively. Kubernetes [21], etcd (a distributed key-value store [4]), and M3 (a distributed time series database and query engine built by Uber [14]) are all implemented in Go. Recently, even an operating system (OS) kernel, Biscuit [52], was implemented in Go [8]. Openstack [18], Paypal [1], Instagram [22], and Dropbox all heavily utilize Python; Python is Dropbox's "most widely used language both for backend services and the desktop client app" with almost 4 million lines of Python code in one repository [20]. As a final example, JavaScript is used in the performance critical path for the Bladerunner pub/sub system at Facebook [49].

Several factors come into play when selecting a programming language for a new service, including current developer expertise and experience, constraints imposed by existing ecosystems (e.g., home-grown libraries, development, debugging tools, performance monitoring and logging systems, etc.), and developer productivity. Managed languages are an attractive proposition, precisely because they offer the promise of higher developer productivity, leading to faster project completion times. The performance of the language is rarely a consideration at the outset, in part because of the belief that performance issues can be addressed later, perhaps through horizontal scaling by simply adding hardware. Some go as far as to claim *"Choosing a language for your application* *simply because it's 'fast' is the ultimate form of premature optimization"* [47].

However, performance will ultimately become a priority as the usage of the service begins to scale and the service becomes too slow or the cost of hardware becomes too high. Developers then begin a large sequence of performance optimizations that can grow into herculean efforts. For example, Twitch.tv and others use tricks and tweaked parameters to meet desired GC performance in Go [5, 19]; project Tungsten in Spark goes as far as to bypass the JVM, to squeeze out performance [23].

But there can come a point where incremental optimizations (requiring much time and effort) no longer suffice and a more radical solution must be considered, namely switching to a "better performing" language. A few examples from industry: Stream abandoned Python for Go, as Python would spend 10ms creating objects from data that Cassandra took 1ms to fetch, noting that *"We've been optimizing Cassandra, PostgreSQL, Redis, etc. for years, but eventually, you reach the limits of the language you're using."* Discord switched from Go to Rust claiming that *"Rust was able to outperform the hyper hand-tuned Go version."* [43]. Performance issues are also cited as the main reason Twitter was forced to switch from Ruby on Rails to Scala and Java [40, 42].

When selecting a new language for performance reasons, the question is: what language? Understanding the performance and scalability implications of a (new) language today is non-trivial, especially for managed languages. This is for several reasons.

First, no empirical studies exist that scientifically compare the different managed languages. The primary source of information available today is the blogosphere containing heated "religious" debates that include tunnel-visioned anecdotes with few rigorous measurements to back up stated claims. For example, while many believe programs written in Java run slower than when written in C/C++ [27], others suggest that Java programs can be faster than C, because the JIT compiler produces faster machine code by leveraging a runtime profile [26]. Similarly, there have been polarized debates with respect to the performance of JavaScript [11,39], Go [43,46] and even Python – for example, sources from Paypal claimed that Python offered superior performance over other languages and reported multiple cases where Python outperformed their C++ and Java counterparts while requiring less code [1]. Similarly, developers reported that Python could outperform both C/C++ and Java when using regular expressions or strings [33–35].

Discussions on *scalability* are even muddier. For example, in a popular blog by the official Node.js Medium account, developers conclude that by being event-driven and asynchronous, JavaScript is *ideal* for scaling to millions of concurrent connections, despite its event loop only executing on a single thread [45]. As another example, while it should be well-known that CPython, the de facto runtime for Python today, uses a global interpreter lock (GIL) that will serialize all concurrent thread executions, Paypal's engineering blog claims that it scales well, and noted that *"Dropbox, Disqus, Eventbrite, Reddit, Twilio, Instagram, Yelp, EVE Online, Second Life, and, yes, eBay and PayPal all have Python scaling stories that prove scale is more than just possible: it's a pattern"* [1].

Second, no benchmark suite is publicly available today that enables a meaningful comparison between different managed languages (and their implementations). Existing benchmark suites target specific languages or applications. Extending these benchmarks to other languages is often impossible; for example, the DaCapo benchmark for Java contains applications such as Eclipse, a full-featured IDE [50]. As a result, any comparison on language runtime performance often compares apples to oranges.

Third, language runtime systems are extremely complex software systems, providing multiple abstractions that all affect performance. For example, a developer must understand the interpreter, possibly multiple JIT compilers (e.g., the OpenJDK JVM contains 4 levels of JIT compilation), a memory management subsystem that performs garbage collect, the behavior of thread libraries, etc.

Finally, there are no helpful, publicly available profiling tools for understanding the overheads of language runtime systems. The language subsystems themselves expose little profiling information on their internals. For example, while it is widely speculated that dynamic type checking adds significant overhead [44], V8/Node.js does not expose any performance counters to report this overhead.

In this paper, we present an in-depth quantitative performance analysis of four of the popular managed languages with their most widely used runtime systems: CPython, OpenJDK, Node.js with the V8 engine for JavaScript, and the reference Go compiler [15, 24, 36, 38]. We compare their performance characteristics with that of C++ on GCC as the baseline. Our focus is primarily on understanding their differences with respect to speed and scalability.[1] We chose these languages not just because of their popularity, but because they represent different designs along the following three dimensions:

- ***Typing.*** JavaScript and Python are dynamically typed, meaning the runtime must determine the type of objects at run time, whereas others are statically typed. This allows us to understand the performance impact of dynamic type checking.

- ***Execution modes.*** Only Go is ahead-of-time compiled. The other runtimes first interpret bytecode, and compile hot functions Just-In-Time (JIT). The exception is CPython that only has an interpreter and no JIT compiler. This enables us to compare three execution modes: native-only (Go and

[1] An analysis of resource usage is left to the Appendix, because the findings have largely already been established by prior studies [63], and we did not need to implement any additional profiling mechanisms.

C++), a combination of interpreter and JIT-compiled native (OpenJDK and V8), and interpreter only (CPython).

- ***Concurrency models.*** V8/Node.js is event driven where event handlers are executed sequentially on a single kernel thread. Similarly, CPython has a global interpreter lock (GIL) so only one thread can execute Python code at a time. Go has its own scheduler and provides *user threads* as "goroutines." Its scheduler decides how many kernel threads to use for the developer's goroutines. OpenJDK's `Thread` is simply a kernel thread.

Contributions

The contributions of the paper are as follows:
Runtime instrumentations. We are the first to make publicly available (as artifacts) instrumentations for the three runtime systems of popular managed languages that are not statically compiled, namely OpenJDK, V8, and CPython. Implementing such instrumentations is challenging given the complexity of these runtimes and the fact that two of them are implemented in assembly and IR. Our instrumentations enable bytecode-level profiling of (1) the execution overhead of any target *bytecode* in the interpreter and (2) the dynamic type-checking overhead in Node.js/V8. The profiling information generated, in turn, can be used to guide optimization efforts at the application level and can enable effectual optimizations. The instrumentations are described in §2.
Benchmark suite. We are the first to make publicly available (as artifacts) six applications suitable for evaluating managed languages; these applications were used to create twelve benchmarks. The applications, which range from micro-benchmarks to real applications, cover a variety of scenarios, differing in compute intensity, memory usage, I/O intensity, relative startup time, and the degree of available concurrency. In particular, we took care to expose the differences in the three major design dimensions mentioned above. Three of the six applications are parallel, and we parallelize them using both multithreading and multiprocessing where applicable. The benchmark suite is called ***LangBench*** and is described in §3. The source code of our instrumented runtimes and LangBench can be found at `https://github.com/topics/langbench`.
Comparative analysis. We quantitatively analyse the performance of the benchmarks in our suite and identify how the individual runtimes improve or hinder performance relative to the respective C++ implementations. Our objective was to compare the runtimes of the target managed languages in an objective, scientific way. Many of our results are not particularly surprising (even if they contradict some views held in the blogosphere). For example, Go and OpenJDK perform significantly better than V8/Node.js and CPython, with CPython performing worst by far, even when compared against V8 and OpenJDK's interpreter-only execution (§6). CPython and V8/Node.js do not benefit from parallelism; in fact, increasing the number of threads systematically decreases performance (§7). A major source of V8's relatively poor performance is its dynamic type checking, even when the JS code only uses primitive types that never change (§6.1).

Perhaps more surprising is the fact that in many cases, the abstractions offered by runtimes can actually lead to speedups over GCC (§8). This contradicts the conventional wisdom that abstraction comes at the expense of performance [74]. OpenJDK outperformed GCC in three of the benchmarks, because the moving garbage collector actually improves cache locality. This leads to the unintuitive behavior that the more frequently GC is performed, the *better* the overall performance. Go abstracts away the usage of kernel threads, reducing the number of context switches and kernel threads. Finally, abstracting away low-level I/O operations allows runtimes to use optimal I/O system call configurations, outperforming the idiomatic approach in C++.

Limitations

The main limitation of our work is that it does not, and cannot, comprehensively answer every question one might have related to the performance of a language runtime. We only evaluated the runtimes of four languages, and for each language we only evaluated the implementation that is the most widely used. In addition, we only ran our workloads on a single OS/hardware stack. Our findings pertain to our benchmarks, which model real-life applications, but may not be representative of a vast range of applications. Accordingly, our study is not meant to determine the best or most performant programming language for any particular application. Furthermore, our benchmark and analysis do not focus on some performance aspects. Notably, we do not study the overhead of garbage collection when under memory pressure (there is a gap between the working set size of our benchmarks and the maximum heap size setting). There is a large body of prior work focusing on this aspect already [51, 63, 79]. Similarly, our benchmark is not meant to measure the various JIT compiler's optimizations, as there are also a large number of existing benchmarks meant to do exactly that [10, 32, 53].

2 Language Runtime Instrumentation

In this section we describe how we instrumented the three runtimes: OpenJDK's HotSpot JVM, Node.js/V8, and CPython. Our instrumentations measure two types of information: (1) the performance of the execution of any bytecode instruction in the interpreter, and (2) the dynamic type and bounds checking overhead in V8's JIT compiled code. Users can specify a bytecode instruction to measure its overhead, or any JavaScript (JS) function to measure the type and bounds checking overhead when executing that function.

Why profile interpreter performance? Some have the view that interpreter performance is not important as it mostly affects the startup time, which will be amortized by "warm execution." We do not share this view. While interpreter performance may have been irrelevant over a decade ago when workloads ran in large, long-running monolithic applications that handle all requests [75], the paradigm shift to the cloud [65, 69, 77] and data analytics [62] expose the runtime's startup performance as being significant. For example, auto-scaling in the cloud often results in the bringing up of additional instances in the face of a load spike [65, 69]; the problem is also exemplified by short-running instances in Function-as-a-Service platforms [65, 77]. In 2020, the median AWS Lambda invocation ran for only *60 milliseconds* [37], while startup times for the JVM and V8 are on the order of hundreds of milliseconds or even seconds [62, 65, 77, 80]. Similarly, data analytics systems face a fundamental tension between parallelizing long running jobs into shorter tasks and the runtime's start-up overhead [62].

In practice, instead of ignoring the performance of the interpreter, implementers spend huge efforts in optimizing the interpreter. For example, OpenJDK has two interpreter implementations: one in C++ and the other entirely in hand-crafted x86 assembly; in one benchmark (§6.2), we found that the C++ interpreter to be 1.93x slower than the assembly one, which is perhaps why the C++ interpreter is only used on non-x86 platforms. Similarly, V8's interpreter is written in hand-crafted IR, and IBM's OpenJ9 Java runtime has significant optimizations targetting startup time which is featured as a major advantage over OpenJDK in the cloud [55, 69, 75].

Bytecode-level profiling can guide optimization efforts and can enable effectual optimizations. For instance, developers can optimize their programs to avoid the use of bytecode instructions with high overheads. Instagram engineers did just this by instrumenting CPython to identify the bytecode instructions with high overheads, and then optimizing their code to avoid using these expensive instructions [22]. Bytecode-level profiling also allows us to understand the performance difference between different runtimes.

Why profile type and bounds checking? As we will show in §6.1, dynamic type and bounds checking is a major source of V8's overhead. Similar to bytecode profiling, programmers can optimize their JS programs to avoid such overhead once the source is identified (§6.1). Our instrumentation also enables eliminating type and bounds checking overhead entirely for those functions where developers know that they are safe. For instance, say a JS function accesses `a[i]`, the element at index `i` of array `a`, and their types never change (known as "monotype"). V8 detects that `a` and `i` are monotype, and it speculatively compiles the function: it checks `a` against the array type (instead of other types) and `i` against integer, before accessing `a[i]`.[2] But to ensure safety, it cannot remove the checks because their types could dynamically change in the future. In that case, the check will fail, forcing the JIT-compiled function to exit and be destroyed, and V8 will re-execute the function in its interpreter before recompiling it.

By disabling the checking logic in V8's JIT compiler for any user-specified JS function, we effectively create a significantly more efficient, albeit unsafe, version of the function. In the above example, developers could enable this feature to turn off the checks when they know `a` and `i` are monotype, so the JIT-compiled code will directly access `a[i]` by indexing into `a` without any checks (effectively turning the JS function into a C function). The difference in execution time of applications with and without checked functions can be significant: e.g., in LangBench's sort benchmark, disabling the checking in V8 results in 8x speedup (§6.1).

Note that we only instrumented V8's JIT compiler for identifying type and bounds checking overheads, but not its interpreter. This is because unlike the JIT compiler that independently compiles the different functions, the interpreter's checking logic is applied to all functions equally, leaving us only with the option of either performing checks in all of an application's functions or none of them. The latter option would likely to be too risky to be useful in practice. We further note that the checking overhead in the interpreter also becomes negligible when compared to the other overhead from the interpreter, whereas their proportion become much more significant in JIT-compiled code.

Instrumentation Implementation. Conceptually the instrumentations we use to profile bytecode execution in the interpreters are simple. We locate the code block in each interpreter that processes a bytecode instruction, and inject instrumentation around it to collect metrics from the x86 CPU performance counters. In practice, however, adding instrumentation is challenging. One challenge is the complexity of the runtimes: JVM, V8, and CPython consist of approximately 1.2M, 1.0M, and 0.9M lines of code respectively, with little documentation. Instrumenting JVM and V8 is even more challenging as their interpreters are not programmed in a high-level language (e.g., C++) as the other runtimes are, but are generated dynamically at startup time.

The HotSpot JVM has two interpreters. Its default interpreter for x86 is written in hand-crafted assembly (known as the "assembly interpreter"). It also has a interpreter written in C++. We instrumented both. Instrumenting the assembly interpreter brings three challenges. First, one needs to locate the code blocks that process the different bytecodes by searching the assembly code. Second, one has to carefully ensure that the instrumented code does not clobber any registers that are used by the interpreter's logic. Finally, HotSpot writes the assembly instructions of the interpreter into memory when it starts up and then jumps to the memory location of the beginning of the interpreter. Hence, we need to use the same mechanism in order to be able to embed our instrumentation logic (written in assembly) into memory.

[2] It also performs other checks as described in §6.1.

```
1   push rax
2   push rcx
3   push rdx
4   rdtscp        ; saves tsc into EDX and EAX registers
5   shlq rdx,32   ; shift tsc's higher 32 bits up in rdx
6   orq rax,rdx   ; or onto rax
7   movq dst,rax  ; output to a scratch register dst
8   pop rdx
9   pop rcx
10  pop rax
```

Figure 1: The sequence of assembly instructions inlined into the processing of each bytecode instruction.

Figure 1 shows the instruction sequence we inject as part of our instrumentation to obtain the CPU's timestamp counter (tsc). Line 1-3 saves the registers values.[3] The `rdtscp` instruction saves the higher and lower 32 bits of tsc into EDX and EAX respectively, i.e., the lower 32 bits of RDX and RAX [68]. It also clears the higher 32 bits of RDX and RAX. Line 5 shifts the higher 32 bits of tsc, stored in EDX, to the higher 32 bits of RDX, and line 6 effectively concatenates the higher and lower 32-bits of tsc, and stores it into RAX. We embed this instruction sequence at both the beginning and the end of the processing of the target bytecode instruction, so that we can measure the latency by computing the difference.[4] Similarly, we use `rdpmc` to read other performance counters, including those for cycle and instruction counts.

Instrumenting V8 is even more challenging. V8's interpreter is written in hand-crafted intermediate representation (IR). When the runtime starts up, the interpreter's binary is generated dynamically from this IR by the same JIT compiler used at run time in V8. This required us to instrument both the IR code of the interpreter and the JIT compiler so that native instrumentation code is injected correctly.

Specifically, for each target bytecode, we had to locate its processing logic in the interpreter's IR and then add a new type of IR node we introduced. We further had to modify the JIT compiler, so that when it encounters this new IR node, it produces the correct assembly instructions that collects the CPU performance counters. This was challenging because there is little documentation describing the internals of V8's interpreter IR or JIT compiler.

One advantage with JVM and V8 is that developers do not need to recompile the runtime when they wish to profile a different bytecode instruction, but only need to restart the runtime. This is because the interpreters are generated dynamically at startup time. Accordingly, we identify which bytecode instruction is to be instrumented at startup time, generate the appropriate instrumentation code so that it is embedded in the interpreter when written to memory (as with JVM) or generated by the JIT compiler (as with V8).

Instrumenting CPython is far more straightforward because it is written in C++. However, the CPython runtime will have to be recompiled whenever profiling is to be enabled or the target bytecode instruction that should be instrumented is changed. In theory, one could, before the execution of each bytecode, check whether the bytecode is one of the specified target bytecodes, and conditionally execute the instrumentation, but this would add too much overhead.

Instrumentation Overhead. Although our instrumentation could incur noteworthy overhead when enabled on frequently executed bytecode instructions, we only used them to measure a specific bytecode instruction, instead of end-to-end runtime. We only count the number of instructions inside of the measurement instructions, not including our instrumented instructions. This is possible as we control the exact assembly instructions generated, and we verify said assembly using objdump, gdb, and outputting the JIT-compiled assembly.[5]

However, our cycle measurements could be skewed by the measuring instructions limiting the processor's out-of-order execution and pipelining capabilities. This is a limitation, and it is extremely difficult to accurately measure this overhead due to the fact that the very act of measuring the cycle count disrupts pipelining (similar to the observer effect in physics).

3 LangBench

Our goal is to compare realistic applications across different languages. We cannot reuse existing benchmarks as they target specific languages, and extending these benchmarks to other languages is infeasible. For example, porting the DaCapo benchmark requires us to implement Eclipse, a full-featured IDE, in four other languages [50]. Therefore, we chose to build 6 applications from the ground up to cover a variety of workloads. We implemented these applications in each of the 5 languages: C++, Go, Java, JavaScript, and Python. From the six applications, we created twelve benchmarks by varying degrees of concurrency, and exploring alternative implementations of the applications.

We made a best-effort attempt at covering a variety of different types of workloads. Our applications range from micro-benchmarks to real world applications, and they stress three major resource usage categories in different ways, being one or more CPU-intensive, memory bound, and I/O bound. Additionally, we implemented parallel versions of the applications where applicable. They also vary from short running to long running ones. The applications and their categories are shown in Table 1.

It was important to ensure that the applications were implemented in a similar and fair manner in each of the five

[3] We have to manually save the registers because we directly inject code into the assembly code, in contrast to injecting assembly code into C where the `__asm__` block saves the registers.

[4] Intel allows tsc to be synchronized across multicore [48], and Linux enables this synchronization [13]. This ensures a meaningful counter value even if the interpreter thread is migrated to another core during the processing of a bytecode instruction. In reality, however, migration is rare given the processing of bytecode instruction typically only takes tens of cycles.

[5] There is no overhead for removing the type and bounds checking in V8 as the compiler only removes instructions.

Application	CPU	Memory	I/O	Parallel
Sudoku solver	✓			
String sorting	✓			
Graph coloring	✓	✓		
Key-Value store		✓	✓	✓
Log analysis	✓	✓	✓	✓
File server			✓	✓

Table 1: The applications and the component(s) they stress.

languages. The design of every implementation of an application is conceptually identical: they use the same algorithms and control flow. Each application is relatively small, so that it could be built in all languages using the same algorithms and data structures. This kept the complexity of the application similar across the languages.

Yet, we fully rewrote the applications in each language, providing our best effort to make the code idiomatic. We referred to official language documentation (e.g. [7]). In certain cases we also implemented different versions of the code. For example, in the JavaScript Sudoku implementation we re-wrote the code multiple times to change the storage of the arrays (see §6.1 for details). For Python and JavaScript we also tried versions that create parallelism (e.g. multiprocessing in Python) and versions that only provide concurrency (e.g. threading in Python). In general, as we analysed each bottleneck, we also tried to find any more performant implementations.

The six applications are:

Sudoku Solver. We implemented an exhaustive search sudoku solver, borrowing from the Spec CPU 2017 benchmark [30]. The algorithm recursively labels all empty cells. At each cell, it verifies the grid state, using the next digit for the cell if verification fails. If all digits are exhausted for a cell, it backtracks to the previous cell.

String Sorting. We implemented the in-place merge sort algorithm described by Katajainen *et al.* [59] and use it to sort strings. First, we permute every possible string of length 6 with 18 possible letters, creating 18^6 strings. These strings are stored in an array, which are then sorted.

Graph Coloring. Graph coloring labels each vertex in a graph, such that no two vertices with an edge between them have the same label. We implemented the algorithm presented by Wigderson [78] which uses a bounded number of colors with run time complexity polynomial in the number of vertices, edges, and the chromatic number. The benchmark colors the YouTube social network and ground-truth communities graph from the Stanford Large Network Dataset Collection [60]. We implemented both a recursive and an iterative version of the algorithm.

Key-Value Store. We implemented an in-memory key-value store based on the general architecture of Redis [28]. We stress the server with Redis' packaged benchmark by running a `SET` test followed by a `GET` test. Each test makes 2 million requests, randomly selecting a key from a space of 500 thousand keys, using a value size of 64 bytes. The Redis benchmark opens a configured number of client connections to the key-value store. Each connection performs an equal number of requests and is treated as a unit of concurrency (such as a thread). The clients are run on a machine separate from the one running the key-value store.

Log Analysis. We implemented the algorithm of CLP that parses logs by separating highly repetitive static text from variable values, and stores them in two different indexes [73]. Logs are queried, returning the matching log messages by searching the index and raw log. We separate the searching into two separate tests. "Regex" searches the raw logs using regular expressions, whereas "Indexed" searches using indexes. Both tests can be run with parallelism, where the files to be searched are partitioned equally. We process 7000 log files totalling 1.21 GB on disk with an average size of 181 KB. The logs were generated by running various jobs from HiBench [56, 57]. Each test first indexes the logs and then performs 7 queries.

File Server. We implemented an HTTP server that serves static files from a directory. A single C++ client is always used that spawns a configured number of threads; each thread connects to the server and requests an equal partition of 1000 real log files with an average size of 16.8 MB. The server implementations handle these connections concurrently, treating each connection as a unit of concurrency. The client and server run on different machines.

4 Methodology

We ran our experiments on two in-house servers, each having 2 Xeon E5-2630V3, 16 virtual cores, 2.4 GHz CPUs, 256 GB DDR4 RAM and two 7200 RPM hard drives. They are running Linux 4.15.0 and connected by a 10 Gbps interconnect. For C++ programs we used GCC 9.3.0 compiling with `-O3` against the C++17 standard. For OpenJDK 13 [17], CPython 3.8.1 [25], and Go 1.14.1 [6], we used the reference implementations for each respective language. We use Node.js 13.12.0 [16] which uses V8 7.9.317.25 [41].

We ran each benchmark 5 times and report the average. The key-value store, log parser, and file server benchmarks were run with the number of both client and worker threads ranging from 1 to 1024. For OpenJDK and V8 the minimum amount of memory was set by determining the first heap configuration that did not cause a crash; for Go, GOGC was set to 5%. We then continuously increased the heap settings until performance no longer improved. We used the results from the first setting (i.e., the smallest heap size) that resulted in optimal performance. For the log parser and file server benchmarks, the used log files were stored on a distributed file system with a replication factor of 2. We cleared Linux's page cache before running each benchmark.

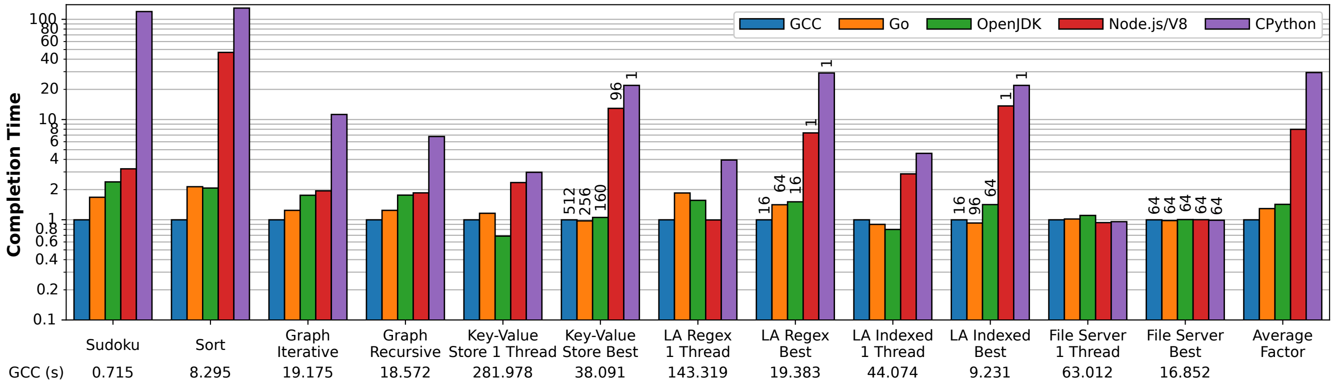


Figure 2: Relative completion times for various language implementations normalized to optimized code under GCC. Note the logarithmic scale of the y axis. "LA" refers to the log analysis application. The numbers at the bottom shows the benchmark's absolute execution time in the C++ implementation. For benchmarks with concurrency, the "Best" bars are annotated with the thread count that results in best completion time. For key-value store and file server it is the number of client threads, not the number of threads used server side. For GCC and OpenJDK, the server creates 1 (kernel) thread to handle each client thread, so the number of server-side threads is the same as the client. For both Node.js and CPython, their best completion time in key-value store is achieved when using a single server-side thread (due to their scalability characterstic described in §7). As for the file server benchmark, both Node.js and CPython's best performance is achieved when using 64 server-side threads (§7). The number of server-side threads in Go is automatically determined by the runtime as described in §8.2. The number of threads for log analysis is the number of worker threads (as there is no client).

5 Overview of Results

Figure 2 shows the run times for the benchmarks in LangBench. Unsurprisingly, optimized GCC was the *fastest* on average, with Go and OpenJDK close behind, being 1.30x and 1.43x slower than GCC. Impressively, Go and OpenJDK outperform optimized GCC for 3 out of the 12 benchmarks. V8/Node.js and CPython performed the worst with run times 8.01x and 29.50x slower than GCC. At the extreme, CPython was 129.66x slower than GCC (for the sort benchmark). V8/Node.js and CPython were competitive with GCC only when the workload is bottlenecked by disk I/O, i.e., in the file server benchmark.

We also found that V8/Node.js and CPython are limited with respect to achievable parallelism. Their design serializes the threads' computation, and requires expensive serialization for different threads (V8) or processes (CPython) to communicate. This leads to the unintuitive result that adding additional threads actually slows down parallel applications as more serialization is required. In fact, for both the key-value store and parallel log analysis benchmark, the best performance is achieved using only a single thread. In contrast, both Go and OpenJDK scale to multicores. Go achieves a 1.02x speedup over GCC in the multithreaded key-value store benchmark, despite being slower in the single threaded version.

In the subsequent sections, we use our instrumented runtimes to provide detailed analyses that explain these results. Specifically, for each runtime, we analyze (at minimum) the two worst performing benchmarks, considering both single-threaded (§6) and multi-threaded (§7) variants for those with parallelism. We further analyzed every case where the runtime outperforms GCC (§8).

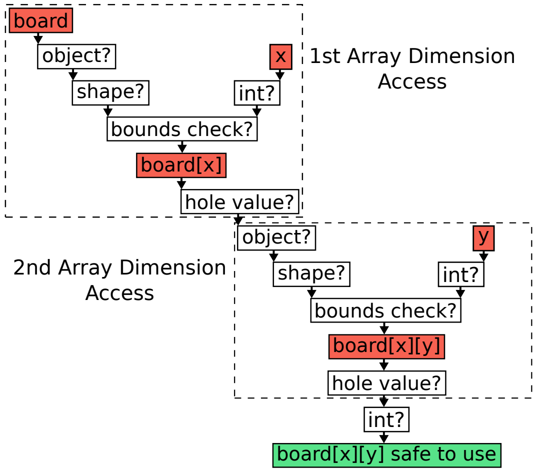


Figure 3: Checks required to access `board[x][y]` in V8/Node.js.

6 Runtime Overhead (Single-thread)

This section investigates the source of runtime overheads on the LangBench single-threaded applications that performed poorly. Specifically, we found (1) type and bounds checking (§6.1) is the bottleneck for V8 in its slowest benchmarks (Sudoku and Sort); (2) interpreter performance (§6.2) is the major cause of CPython's overhead – despite lacking a JIT compiler, its interpreter performs much worse compared to OpenJDK and V8; (3) GC write barrier (§6.3) is the bottleneck in the Sort benchmark for both OpenJDK and Go, which is their slowest workload, even when heap usage is small.

6.1 Type and Bounds Checking Overhead

We found that type checking and bounds checking made up 41.83% and 87.43% of V8's execution time in the default su-

	Code Version	Time (s)	Overhead of Checks (%)
	Default	2.369	–
1-2	Remove Obj./Int Checks	2.177	8.105
3	Remove Shape Check	2.219	6.332
4	Remove Bounds Check	2.154	9.076
5	Remove Hole Check	2.051	13.423
1-5	Remove All Checks	1.378	41.832

Table 2: We modified V8's JIT compiler and removed each of the checks performed for a 2D array access to `board[x][y]` shown in Figure 3. We measured the resulting execution time, and compare it against the default execution time with all checks. We also show the execution time when all checks are removed.

doku and sort benchmarks, which are the two single-threaded benchmarks where V8 showed the worst performance compared to GCC. Note that V8 has other sources of overhead, such as execution being partially interpreted, when compared with GCC. For the numbers in this sub-section, we compare against the default execution time when the runtime binary is in Linux's page cache, unlike the results in Figure 2, where we clear the page cache before each test. Next we zoom into the Sudoku benchmark to explain this overhead, and how we can leverage our bytecode profiling information to optimize our JavaScript code.

For V8/Node.js, Sudoku spends 93% of its time primarily comparing 2D array elements of the sudoku board. The majority of this time is spent performing 11 type and bound checks for each 2D array access, as shown in Figure 3. Each dimension requires 5 checks, and the 11th check is used for the final value. Table 2 shows the overhead for these checks.

The first check ensures that `board` is an object pointer, by checking for a tagged bit to distinguish between an object pointer and a primitive integer value. (V8 stores both object pointers and primitive integers in a 8 byte word, so that integers can be stored inline instead of being allocated on the heap.) Second, V8 must similarly check that `x` is an integer, rather than an object. Omitting these checks made it 8.1% faster (shown in Table 2) — removing them is safe, as we know that no incorrect type will be used.

After V8 confirms that `board` is an object, it checks that the internal type of `board`, called a shape, is an array. Fourth, V8 performs a bounds check for the access to `board[x]`. Finally, V8 checks if the value accessed is a "hole". In JavaScript, arrays may be sparse, meaning not every index has a value. Indexes without values are called holes, which V8 must convert to `undefined` if accessed. The same checks must be repeated to access the second dimension of `board`. To use `board[x][y]`, a last check is necessary to verify it is an integer.

Profiling enabled optimization. Initially, we preallocated the fix-size sudoku board. In V8, preallocated arrays are created sparse as their values are uninitialized, requiring the hole checks. Even though the array was filled with integers before being used, sparse arrays never lose their status.

```
// OpenJDK: bounds check board[x].length > 8
for (int i = 0; i < 9; i++) {
    // Go: bounds check board[x].length > i
    int element = board[x][i];
    ...
}
```

Figure 4: Code showing where Go and OpenJDK perform array bounds checking when accessing `board[x][i]` in a loop.

Bytecode		Insn. per BC	Cycles per BC
OpenJDK	Assembly `aaload`	12	7.7
	Assembly `iaload`	11	7.1
	C++ `aaload`	33	12.5
	C++ `iaload`	22	11.1
Node.js	`LdaKeyedProperty`	90	26.3
CPython	`BINARY_SUBSCR`	138	41.8

Table 3: Statistics for array access bytecodes (BC) performed by various interpreters for the sudoku benchmark.

We implemented an optimized version which would create arrays without holes, known as "packed" arrays. This optimized version was 1.48x faster (and is what is shown in Fig. 2). Our optimized sudoku benchmark for V8/Node.js starts with an empty array, then appends 9 `Int8Array`s to create the 2D sudoku board. This allows V8 to recognize that there are no holes. Using the built in `Int8Array`, preallocation initialized it with the default value of 0, rather than a hole.

Unfortunately, these optimizations cannot be applied universally. First, it presents a trade-off that can only be determined via profiling: while sparse arrays require hole checking, building a large packed array requires many internal resizing operations to grow the array. In addition, typed arrays such as `Int8Array` only exist for certain integral types. For example, it is not possible to preallocate a packed array of strings or any user defined type.

GCC, Go, and OpenJDK. Compared to GCC, Go and OpenJDK must also perform similar bounds checks. However, they avoid the type checking as they are statically typed.

OpenJDK successfully lifts all loop-invariant computations to outside the loop. In the code of Figure 4, OpenJDK determines that the maximum value of `i` used to access `board[x][i]` is 8, and checks if the length of `board[x]` is greater than 8 outside the loop. Go performs the bounds check in each iteration. Further, 2D arrays in OpenJDK contain pointers to 1D arrays, which may be null. However, OpenJDK has an optimization which eliminates null checks, and instead catches them using a signal handler for `SIGSEGV`. On the other hand, Go does not need to perform null checks as its 2D arrays are laid out contiguously in memory like in C.

6.2 Interpreter Overhead

CPython is slower than the other runtimes because it lacks a JIT compiler and so programs are strictly interpreted. Therefore we further compare the three runtimes by running sudoku on each of them only in interpreter mode. OpenJDK's (assembly) interpreter outperforms both V8 and CPython by 2.59x and 5.34x respectively. This is because static typing allows OpenJDK to avoid the type checks that V8 and CPython must perform. OpenJDK has dedicated bytecodes for accessing different types of arrays (`aaload` for an array of arrays, `iaload` for an integer array). In contrast, V8 and CPython both have a single bytecode (`LdaKeyedProperty` and `BINARY_SUBSCR`, respectively) which must accommodate for any array or dictionary type. Table 3 shows the performance profiling results of different bytecode executions, using our instrumentations.

CPython is still 2.07x slower than V8, even though both of them do dynamic typechecking. As shown in Table 3, CPython uses 138 instructions and 41.8 cycles to execute each byte code instruction (`BINARY_SUBSCR`), whereas Node.js only spends 90 instructions/26.3 cycles to process each byte code instruction (`LdaKeyedProperty`). This is due to the optimizations of V8's interpreter: it is hand-crafted in IR, whereas CPython is implemented in C. Similarly, we found that OpenJDK's assembly interpreter is 1.93x faster than the one implemented in C++.

We found the hand-crafted interpreter implementation made a few notable optimzations. First, it aggressively inlined functions. The CPython bytecode we inspected ended up containing around 5-6 function calls in the common execution path. The equivalent bytecode in Node.js had no call instructions, similar to when all functions are completely inlined. This further enables more aggressive optimization. For example, error handling logic that would be functions in C code can now be grouped together at the end. This leaves the instructions in the non-error paths tightly together and improves cache performance. In addition, developers have a better understanding than the compiler on what the common path is, so that they can manually group the basic blocks that are commonly executed together (and move error handling logic to the end).

Theoretically GCC could also perform the same level of inlining, and developers can manually use `goto` statements to move all error handling logic to after the common-path logic. However, performing aggressive inlining unselectively could hurt performance (increased function size, register pressures, etc.), and excessive use of `goto` could hurt the readability, reliability, and maintainability of the software.

Performance Sensitivity on Interpreters. We observe that an interpreter may amplify the performance overhead caused by small code changes that would only incur negligible overhead in compiled execution. Under CPython, the iterative version of graph coloring ran 1.66x slower than the recursive version, in contrast to the other interpreters. The function performing the iterative algorithm contained 1.54x more bytecodes than the recursive function, resulting in 22% more instructions recorded by `perf`. In contrast, for GCC, switching from recursive to iterative adds only 4% more instructions.

Iterative versions of recursive functions are commonly necessary to avoid stack overflows. Instead of a recursive function call, the iterative function appends the call arguments to an array, and later pops the arguments off the array to perform another iteration of the algorithm. In addition, the iterative algorithm must check if the array is empty at each iteration of the loop. These seemingly simple operations significantly increase the bytecode count and execution time. JIT compilers mitigate these extra operations through optimizations such as reducing the number of redundant checks.

Startup Overhead for OpenJDK and V8. OpenJDK and V8 spend 843ms and 788ms, respectively, in startup in the Sudoku benchmark. Startup is the primary reason for OpenJDK being slower than GCC when running Sudoku, as it is the shortest benchmark. Specifically, 708ms is spent loading the JVM's large binary from disk, while the rest (135ms) is spent in classloading and interpreter execution. In comparison, OpenJDK's warm execution time is only 868ms (when we run the Sudoku benchmark in a JVM that has already been warmed up by running the same benchmark multiple times), whereas GCC's warm execution time is 611ms.

6.3 GC Write Barriers

We were surprised to see that under OpenJDK's default GC setting, it was 10.03x slower than GCC for the Sort benchmark. Sort is also the benchmark where Go performs the worst relative to GCC: 2.14x slower. The source of the slowdown for both OpenJDK and Go is the cost of GC write barriers. This cost occurs despite GC hardly ever running in Sort, as write barriers are necessary to maintain data structures needed to perform GC. Interestingly, for OpenJDK, using the non-default Parallel GC algorithm drops the slowdown to only 2.07x (shown in Figure 2), as it contains fewer instructions. Go's write barrier contains even fewer instructions, and is slightly faster than OpenJDK with Parallel GC.

Write barriers bring a constant cost to pointer writes regardless of how often GC is actually performed. For our in-place merge sort, swapping two elements is the primary source of write barriers. This requires two write barriers, one for each element being written. OpenJDK's default GC algorithm, G1 [12], adds 44 instructions for these write barriers, completely dwarfing the 6 instructions required to swap the elements and 5 for bounds checking. On the other hand, Parallel GC's write barriers only use 5 instructions, 8.8x fewer than G1.

Both Go and G1 require a write barrier to ensure every live object is captured when they perform marking concurrently with application threads. Furthermore, both G1 and Parallel GC in OpenJDK divide the heap into regions and move live objects across regions to compact the heap. For both, write barriers are used to maintain remembered sets, which are used to find and update pointers to moved objects. However, G1 performs more checks during its write barrier to avoid unnecessary updates to the remembered sets. This avoids work when using the remembered sets to update pointers, and helps minimize pause time.

7 Scalability Limitations

We found that CPython and Node.js limit the degree of parallelism achievable. We first briefly introduce the background of the Node.js and CPython concurrency model and then describe our findings.

Background. Node.js is event driven; by default, it uses a single Node.js thread to drive an event loop and process all incoming events. If the processing of an event blocks (e.g., on I/O), the underlying kernel thread will block, and Node.js's event loop continues with another kernel thread to process the next event. In other words, multiple threads can be blocked at the same time, but CPU execution is serialized. While Node.js supports running multiple Node.js threads (known as worker threads), each runs its own event loop. Hence worker threads *do not* share the heap (to avoid data races); data sharing requires message passing with data being serialized.

In essence, CPython's concurrency model is the same as that of Node.js where multiple kernel threads can block on I/O at the same time, except that it is the programmer's job to create the threads; the threads share the same heap. In CPython's case, CPU computation is serialized by the Global Interpreter Lock (GIL) so that only one thread can use the CPU at a time. CPython also supports `multiprocessing`, forking different processes to avoid the GIL. However, data sharing and communication requires serialization.

Node.js and CPython's scalability on LangBench. We can now explain the scalability patterns of Node.js and CPython. We ran three parallel benchmarks, namely log analysis, key-value store, and file server, under different configurations, including different number of threads, as well as parallelizing them with multiple processes in CPython. In log analysis and key-value store, *the best performance is achieved using a single CPython or Node.js thread*, whereas the other runtimes are able to improve performance by adding more threads.

These two benchmarks, namely log analysis and key-value store, are bottlenecked by CPU or memory accesses, instead of blocking I/O. Therefore, creating multiple threads offers no advantage in Node.js and CPython as their executions are serialized. In the case of Node.js, performance degrades significantly when creating additional worker threads due to the serialization overhead. On indexed search log analysis, Node.js's performance drops 4.7x when we use more than one worker thread. In this benchmark, multiple workers communicate frequently as they share the same dictionaries. Similarly, serialization overhead slows down CPython when we switch to multiprocess, resulting in a 4.9x slowdown on the same benchmark. While multiple CPython threads share the heap, they still introduce thread management overhead compared to using a single thread.

Specifically, in key-value store, CPython can only scale to one client thread (adding additional concurrent client threads will worsen the completion time). In comparison, Node.js/V8 scales up to 96 client threads, even though it only uses 1 Node.js event-loop thread at server side. However, its completion time can not keep improving with more client threads, whereas it still can under GCC, Go, and OpenJDK. Note that the improvement plateaus when the client thread count increases to 160. Even though GCC achieved its best completion time on 512 client threads, the improvement over 160 threads is negligible. This is why in Figure 2, the difference in best completion times is small between GCC, Go, and OpenJDK, even though they are achieved on 512, 256, and 160 client threads, respectively.

In comparison, Node.js and CPython scale well on the file server benchmark. This benchmark is I/O parallel: there is little communication between different threads, and they are bottlenecked by disk I/O. Creating multiple threads (or processes in CPython) thus improves the performance (when there are concurrent client connections).

8 Runtime Advantages

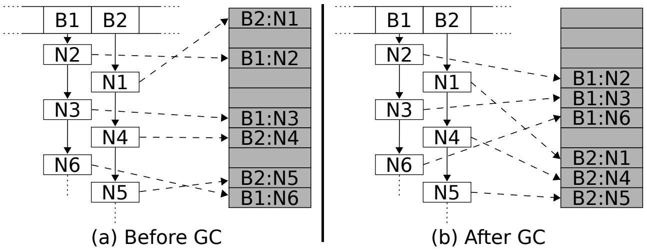
We found that the high-level abstractions provided by the runtimes can, in some cases, result in better performance and scalability. This is counter-intuitive given the conventional wisdom that abstractions generally come at the expense of performance [74]. We discuss three findings: (1) object relocations in OpenJDK's moving GC can result in better cache locality; (2) Go's scheduler automatically maps user threads to kernel threads, and hence abstracts away the direct usage of kernel threads, reducing the number of context switches and the number of kernel threads used; (3) abstracting away the low-level I/O operations allows runtimes to use the optimal I/O system call configurations.[6]

8.1 GC Improved Cache Locality

OpenJDK's moving garbage collector can significantly improve cache locality, resulting in speedups in three benchmarks: single threaded key-value store and both iterative and recursive implementations graph coloring. In particular, OpenJDK was much faster than GCC at the single threaded key-value store, with 1.46x speedup. This is the largest speedup any runtime had over GCC.

Key-value store. We found that the source of cache locality was from iterating over linked lists. Our key-value store implements a hashtable with separate chaining, meaning hash

[6] There are a few more cases where runtimes demonstrated better performance than GCC; they are related to the implementation of libraries. We discuss them in detail in the Appendix.



(a) Before GC

(b) After GC

Figure 5: The key-value store before and after a GC pause. White boxes logically represent Java objects, and the shaded boxes represent the objects' location in the JVM heap. A 'B' denotes a bucket mapped to by the hash function, and an 'N' denotes a node in the bucket's linked list. The number of the node represents the order they are inserted into the hashtable. The memory for the nodes of the bucket begins scattered, but after GC relocation is ordered by the traversal of the bucket's linked lists.

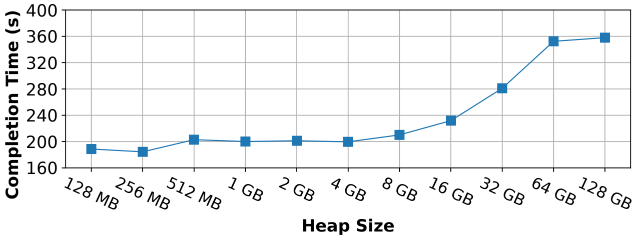


Figure 6: The OpenJDK single threaded key-value store benchmark run with increasing heap sizes, corresponding to fewer GC cycles.

collisions are added to a bucket by appending the key-value (KV) pair to a list. This is shown as the white boxes in Figure 5. For example, N2, N3, and N6 are different KV pairs hashed to the same bucket B1.

OpenJDK uses bump pointer allocation. Therefore, nodes in the hashtable are laid out sequentially in memory based on their insertion order. Figure 5 (a) shows how the nodes of a bucket would be initially laid out in memory. There is little locality, as adjacent nodes of the same linked list are scattered. Therefore, whenever there is a lookup, insertion, or a deletion of a key in the linked list, the traversal of the linked list is expensive due to poor locality.

However, OpenJDK's moving GC reorders the objects in memory. It scans for all live objects that are reachable from the GC roots (e.g., objects on the stack) by following the pointers, copying them to a different memory region, before freeing the old region. For the linked list, this means that the objects will be allocated adjacently, in the same order as in the linked list, as shown in Figure 5 (b).

In comparison, GCC uses a size segregated allocator (`malloc`). Since nodes have the same size, they will be placed in the same region, resulting in a similar pattern as with bump pointer allocation, with nodes laid out in insertion order. When profiling the iteration, we found that GCC actually executed fewer instructions than OpenJDK, but was still slower. In the tight loop iteration, the bucket GCC took only 5 assembly instructions compared to OpenJDK's 11.

This behavior presents the unintuitive case where the more frequently GC is performed, the better the performance. Figure 6 shows that with more frequent GC cycles, objects are re-ordered in memory more often, leading to improved performance. We control the frequency of GC by using different heap sizes. The larger the heap, the fewer GC cycles. When it is 128 GB, performance is the *worst* because GC is never triggered; objects are never moved, so there is no locality.

To verify that cache locality was the source of the performance gap, we modified OpenJDK to expose a method to print the virtual address a reference points to. We do this as GC obfuscates `perf` cache hit rates, making them impractical to compare. In one run where no GC was performed, over 99% of the distances between nodes of the linked lists were different, with a median distance of 724 KB. A run with GC was 1.86x faster; 57% of nodes were 88 bytes apart, and 41% were 192 bytes apart. Although the size of a cache line on our processors is 64 bytes, so two nodes would not be in the same cache line, it is likely that they are in adjacent cache lines, opening the opportunity for prefetching.

Graph coloring. We found OpenJDK outperformed GCC (by 1.37x) on graph coloring, when the C++ program uses the standard library. Our investigation showed that GC had a similar effect as for the key-value benchmark given that graph coloring also uses a hash table. Both hash table implementations on OpenJDK (`HashMap` and `HashSet`) and C++'s standard libraries (`std::unordered_set` and `std::unordered_map`) use an open hashing design; i.e., it uses separate chaining to connect the elements in a linked list upon collision. As a result, both GCC and OpenJDK suffer from poor locality initially. However, OpenJDK quickly gains locality through GC, as with the key-value store benchmark.[7]

8.2 Scalability in Go

In the multithreaded key-value store implementation, Go has a 1.02x speedup compared to GCC, despite being 1.16x slower than GCC in the single threaded version. Go outperforms GCC by avoiding 2.2 million context switches through the use of asynchronous networking I/O and significantly fewer kernel threads. With GCC, network I/O is performed using synchronous system calls, blocking the kernel thread, resulting in a context switch. When goroutines perform I/O, the work is offloaded to an internal goroutine which uses asynchronous system calls. A goroutine performing I/O is blocked by Go's scheduler, but the underlying kernel thread is not blocked; instead, Go schedules another goroutine on the same kernel thread. As a result, Go only uses at most 42 kernel threads, regardless of the number of concurrent client threads. (The number of kernel threads is automatically chosen by the Go runtime depending on the workload's characteristic.)

[7] We optimized our C++ benchmark by switching to hashtable implementations from Google's Abseil library [2], which uses a closed hashing implementation that achieves better locality.

We verified that context switching causes the majority of the 600 ms gap between the fastest multithreaded Go and GCC execution times. Using LEBench [72], we measured the average cost of a context switch on our machine to be 5.84 μs. With 32 cores, `perf` reports approximately 70K context switches per core, which adds up to 409ms of overhead, making up the majority of the 600ms performance gap.

8.3 I/O System Calls in the File Server

To read a file in the file server benchmark in C++, we initially used the more general, idiomatic approach which uses iterators. This results in repeated fixed size `read` system calls. Unlike C++, all the managed runtimes abstract away the low-level system call interfaces when performing I/O, so that they can transparently issue system calls in an optimal way, by first calling `fstat` to get the file size, followed by *a single* `read` for its entire contents. All runtimes use this approach when reading a file. So any developer using the runtimes will benefit from the optimizations without any burden of knowledge. In comparison, we have to manually optimize our C++ implementation to switch to `fstat` and `read`, leading to a 2x speedup.

9 Related Work

Ours is the first performance study to analyze and compare the implementations of multiple widely used runtimes, and provide the necessary instrumentations to do so. There are existing benchmarks to evaluate software performance, but they focus on novel benchmark methodologies. Marr *et al.* designed a benchmark suite with the goal of having a methodology for evenly comparing a common subset of language abstractions [64]. They limit their applications to a minimal set of primitive operations and exclude built-in data structures such as hashtables to ensure that no language has an advantage. Rather than strictly stressing the compiler on specific primitive operations, we evaluated all aspects of a runtime on how they affect performance under different scenarios using idiomatic code. Both DaCapo [50] and Renaissance [70] created benchmark suites consisting only of Java applications for various workloads. TailBench created a statistically sound methodology for measuring latency-critical applications in C++ and Java [58]. SPEC [31] and the Computer Language Benchmarks Game [3] provide a variety of benchmarks, covering many languages, but present no analysis. In contrast, our work focuses on understanding and providing an explanation for the technical details of language implementations that cause performance differences.

Other studies of languages have had different scopes, focusing tightly on a specific language or aspect. By utilizing the Rosetta Code [29] repository, Nanz *et al.* present statistical findings, such as the fact that scripting languages are more concise than procedural languages [67]. Nanz *et al.* further studied the usability and performance of Chapel, Cilk, and Go in multicore workloads [66]. Prokopski *et al.* study interpreter code-copying optimizations in the SableVM, OCaml, and Yarv interpreters [71]. Wade *et al.* quantify the impact of profile data on JIT compiled code quality in the HotSpot VM.

Lion *et al.* instrumented the JVM to measure startup times (i.e., the total time spent in class loading and interpreter) [62]. However, they did not provide fine-grained instrumentation to profile the execution of each bytecode instruction.

10 Concluding Remarks

We presented an in-depth performance analysis of runtimes under a variety of scenarios. We implemented LangBench, a benchmark suite that enables an objective comparison of language implementation performance. Our runtime instrumentations facilitate understanding *why* a runtime performs well or poorly. We demonstrated that our instrumentations provide valuable profiling information that enables optimizations. We have open-sourced our instrumentations and LangBench so that practitioners can use and enhance them to analyze and optimize their applications.

Acknowledgements

We thank the anonymous reviewers and the shepherd for their insightful comments. This research was supported by the Canada Research Chair fund, an NSERC Discovery grant, and a VMware gift.

References

[1] 10 Myths of Enterprise Python. https://medium.com/paypal-engineering/10-myths-of-enterprise-python-8302b8f21f82.

[2] Abseil. https://abseil.io/.

[3] The Computer Language Benchmarks Game. https://benchmarksgame-team.pages.debian.net/benchmarksgame/.

[4] etcd - A distributed, reliable key-value store for the most critical data of a distributed system. https://etcd.io/.

[5] Go memory ballast: How I learnt to stop worrying and love the heap. https://blog.twitch.tv/en/2019/04/10/go-memory-ballast-how-i-learnt-to-stop-worrying-and-love-the-heap-26c2462549a2/.

[6] The Go Programming Language. https://golang.org/.

[7] Go Programming Language Documentation. https://go.dev/doc/.

[8] Go Programming Language Specification. https://golang.org/ref/spec.

[9] Introduction to Intel Advanced Vector Extensions. https://software.intel.com/content/www/us/en/develop/articles/introduction-to-intel-advanced-vector-extensions.html.

[10] Java EE: DayTrader Benchmark. https://github.com/OpenLiberty/sample.daytrader8.

[11] JavaScript is slow. https://kariera.future-processing.pl/blog/javascript-is-slow/.

[12] JEP 248: Make G1 the Default Garbage Collector. http://openjdk.java.net/jeps/248.

[13] Linux source tsc_sync.c: Check tsc synchronization. https://github.com/torvalds/linux/blob/df0cc57e057f18e44dac8e6c18aba47ab53202f9/arch/x86/kernel/tsc_sync.c.

[14] M3: Uber's Open Source, Large-scale Metrics Platform for Prometheus. https://eng.uber.com/m3/.

[15] Most popular languages on GitHub. https://github.com/oprogramador/github-languages.

[16] Node.js. https://nodejs.org/en/.

[17] OpenJDK 13. https://openjdk.java.net/projects/jdk/13/.

[18] OpenStack Overview. https://www.openstack.org/software/.

[19] Optimizing a Golang service to reduce over 40% CPU. https://coralogix.com/log-analytics-blog/optimizing-a-golang-service-to-reduce-over-40-cpu/.

[20] Our journey to type checking 4 million lines of Python. https://blogs.dropbox.com/tech/2019/09/our-journey-to-type-checking-4-million-lines-of-python/.

[21] Production-Grade Container Orchestration - Kubernetes. https://kubernetes.io/.

[22] Profiling CPython at Instagram. https://instagram-engineering.com/profiling-cpython-at-instagram-89d4cbeeb898.

[23] Project Tungsten: Bringing Apache Spark Closer to Bare Metal. https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html.

[24] PYPL PopularitY of Programming Language. http://pypl.github.io/PYPL.html.

[25] Python Implementations - Python Wiki. https://wiki.python.org/moin/PythonImplementations.

[26] Quora: In what cases is Java faster than C. https://www.quora.com/In-what-cases-is-Java-faster-if-at-all-than-C.

[27] Quora: In what cases is Java slower than C by a big margin. https://www.quora.com/In-what-cases-is-Java-slower-than-C-by-a-big-margin.

[28] Redis. https://redis.io.

[29] Rosetta Code. https://rosettacode.org/wiki/Rosetta_Code.

[30] SPEC CPU 2017 Documentation. https://www.spec.org/cpu2017/Docs/#benchmarks.

[31] SPEC: Standard Performance Evaluation Corporation. https://www.spec.org.

[32] SPECjbb 2015 Benchmark. https://www.spec.org/jbb2015/.

[33] Stack Overflow: C++11 regex slower than python. https://stackoverflow.com/questions/14205096/c11-regex-slower-than-python.

[34] Stack Overflow: Why do std::string operations perform poorly? https://stackoverflow.com/questions/8310039/why-do-stdstring-operations-perform-poorly.

[35] Stack Overflow: Why is python faster than c++ in this case? https://stackoverflow.com/questions/24895881/why-is-python-faster-than-c-in-this-case.

[36] The State of Developer Ecosystem 2019. https://www.jetbrains.com/lp/devecosystem-2019/.

[37] The State of Serverless. https://www.datadoghq.com/state-of-serverless/.

[38] The State of the Octoverse. https://octoverse.github.com.

[39] Transducers Speed Up JavaScript Arrays. https://itnext.io/using-transducers-to-speed-up-javascript-arrays-92677d000096.

[40] Twitter Shifting More Code to JVM, Citing Performance and Encapsulation As Primary Drivers. https://www.infoq.com/articles/twitter-java-use/.

[41] V8 JavaScript engine. https://v8.dev/.

[42] Why did Twitter switch from Ruby on Rails? https://medium.com/@mittalyashu/why-did-twitter-switch-from-ruby-on-rails-dac66150044d.

[43] Why Discord is switching from Go to Rust. https://blog.discord.com/why-discord-is-switching-from-go-to-rust-a190bbca2b1f.

[44] Why is Dynamic Type Checking Expensive? https://stackoverflow.com/questions/41622341/why-is-type-checking-expensive.

[45] Why the Hell Would You Use Node.js. https://medium.com/the-node-js-collection/why-the-hell-would-you-use-node-js-4b053b94ab8e.

[46] Why we switched from Python to Go. https://getstream.io/blog/switched-python-go/#reason-performance.

[47] Yes, Python is Slow, and I Don't Care. https://medium.com/pyslackers/yes-python-is-slow-and-i-dont-care-13763980b5a1.

[48] Intel® 64 and IA-32 architectures software developer's manual, Volume 3B: System programming guide, part 2, Section 17.15. https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3b-part-2-manual.pdf, 2016.

[49] Jeffrey Barber, Ximing Yu, Laney Kuenzel Zamore, Jerry Lin, Vahid Jazayeri, Shie Erlich, Tony Savor, and Michael Stumm. Bladerunner: Stream processing at scale for a live view of backend data mutations at the edge. In *Proc. 28th ACM Symp. on Operating Principles (SOSP'21)*, page 708–723. Association for Computing Machinery, October 2021.

[50] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The dacapo benchmarks: Java benchmarking development and analysis. In *Proc. 21st Conf. on Object-oriented Programming Systems, Languages, and Applications (OOPSLA'06)*, pages 169–190. ACM, 2006.

[51] Rodrigo Bruno, Paulo Ferreira, Ruslan Synytsky, Tetiana Fydorenchyk, Jia Rao, Hang Huang, and Song Wu. Dynamic vertical memory scalability for OpenJDK cloud applications. In *Proc. Intl. Symp. on Memory Management (ISMM'18)*, pages 59–70. ACM, 2018.

[52] Cody Cutler, M. Frans Kaashoek, and Robert T. Morris. The benefits and costs of writing a POSIX kernel in a high-level language. In *Proc. 13th Symp. on Operating Systems Design and Implementation (OSDI'18)*, pages 89–105. USENIX Association, October 2018.

[53] Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically rigorous Java performance evaluation. In *Proc. 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'07)*, page 57–76. Association for Computing Machinery, 2007.

[54] Hadoop. https://hadoop.apache.org.

[55] Handra. Comparing Hotspot and OpenJ9. https://www.linkedin.com/pulse/comparing-hotspot-openj9-handra-/.

[56] Shengsheng Huang, Jie Huang, Jinquan Dai, Tao Xie, and Bo Huang. The HiBench benchmark suite: Characterization of the MapReduce-based data analysis. In *New Frontiers in Information and Software as Services*, pages 209–228. Springer, 2011.

[57] Shengsheng Huang, Jie Huang, Yan Liu, Lan Yi, and Jinquan Dai. HiBench: A representative and comprehensive Hadoop benchmark suite. In *Proc. ICDE Workshops*, ICDEW '16. IEEE Press, 2010.

[58] H. Kasture and D. Sanchez. Tailbench: a benchmark suite and evaluation methodology for latency-critical applications. In *Proc. IEEE Intl. Symp. on Workload Characterization (IISWC'16)*, pages 1–10. IEEE Press, Sep. 2016.

[59] Jyrki Katajainen, Tomi Pasanen, and Jukka Teuhola. Practical in-place mergesort. *Nordic J. of Computing*, 3(1):27–40, March 1996.

[60] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data, June 2014.

[61] Tim Lindholm, Frank Yellin, Gilad Bracha, Alex Buckley, and Daniel Smith. The Java®Virtual Machine Specification - Java SE 13 Edition. https://docs.oracle.com/javase/specs/jvms/se13/html/.

[62] David Lion, Adrian Chiu, Hailong Sun, Xin Zhuang, Nikola Grcevski, and Ding Yuan. Don't get caught in the cold, warm-up your JVM: Understand and eliminate JVM warm-up overhead in data-parallel systems. In *Proc. 12th Symp. on Operating Systems Design and Implementation (OSDI'16)*, pages 383–400. USENIX Association, November 2016.

[63] David Lion, Adrian Chiu, and Ding Yuan. M3: End-to-end memory management in elastic system software stacks. In *Proc. 16th European Conf. on Computer Systems (EUROSYS'21)*, page 507–522. Association for Computing Machinery, 2021.

[64] Stefan Marr, Benoit Daloze, and Hanspeter Mössenböck. Cross-language compiler benchmarking: Are we fast yet? In *Proc. 12th Symp. on Dynamic Languages (DLS'16)*, pages 120–131. Association for Computing Machinery, 2016.

[65] Colt McAnlis. Improving cloud function cold ctart time, Google Cloud Performance Atlas. https://medium.com/@duhroach/improving-cloud-function-cold-start-time-2eb6f5700f6.

[66] S. Nanz, S. West, K. S. d. Silveira, and B. Meyer. Benchmarking usability and performance of multicore languages. In *Proc. Intl. Symp. on Empirical Software Engineering and Measurement*, pages 183–192. IEEE Press, 2013.

[67] Sebastian Nanz and Carlo A. Furia. A comparative study of programming languages in Rosetta code. In *Proc. 37th Intl. Conf. on Software Engineering (ICSE'15)*, page 778–788. IEEE Press, 2015.

[68] Gabriele Paoloni. How to benchmark code execution times on Intel IA-32 and IA-64 instruction set architectures. Intel Coporation, 2010.

[69] Marius Pirvu. Optimize JVM start-up with Eclipse OpenJ9. https://developer.ibm.com/articles/optimize-jvm-startup-with-eclipse-openjj9/.

[70] Aleksandar Prokopec, Andrea Rosà, David Leopoldseder, Gilles Duboscq, Petr Tůma, Martin Studener, Lubomír Bulej, Yudi Zheng, Alex Villazón, Doug Simon, Thomas Würthinger, and Walter Binder. Renaissance: Benchmarking suite for parallel applications on the JVM. In *Proc. 40th Conf. on Programming Language Design and Implementation (PLDI'19)*, pages 31–47. Association for Computing Machinery, 2019.

[71] Gregory B. Prokopski and Clark Verbrugge. Analyzing the performance of code-copying virtual machines. In *Proc. 23rd Conf. on Object-Oriented Programming Systems Languages and Applications (OOPSLA'08)*, page 403–422. Association for Computing Machinery, 2008.

[72] Xiang (Jenny) Ren, Kirk Rodrigues, Luyuan Chen, Camilo Vega, Michael Stumm, and Ding Yuan. An analysis of performance evolution of Linux's core operations. In *Proc. 27th ACM Symp. on Operating Systems Principles (SOSP'19)*, page 554–569. Association for Computing Machinery, 2019.

[73] Kirk Rodrigues, Yu Luo, and Ding Yuan. CLP: Efficient and scalable search on compressed text logs. In *Proc. 15th Symp. on Operating Systems Design and Implementation (OSDI'21)*, pages 183–198. USENIX Association, July 2021.

[74] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end Arguments in System Design. *ACM Trans. Comput. Syst.*, 2(4):277–288, November 1984.

[75] Hang Shao, Marius Pirvu, Tobi Ajila, and Vijay Sundaresan. Innovations for Java running in containers. https://blog.openj9.org/2021/06/15/innovations-for-java-running-in-containers/.

[76] Spark. http://spark.apache.org.

[77] Kai-Ting Amy Wang, Rayson Ho, and Peng Wu. Replayable execution optimized for page sharing for a managed runtime environment. In *Proc. 14th European Conf. on Computer Systems (EUROSYS'19)*. Association for Computing Machinery, 2019.

[78] Avi Wigderson. Improving the performance guarantee for approximate graph coloring. *J. ACM*, 30(4):729–735, October 1983.

[79] Ting Yang, Emery D. Berger, Scott F. Kaplan, and J. Eliot B. Moss. CRAMM: Virtual memory support for garbage-collected applications. In *Proc. 7th Symp. on Operating Systems Design and Implementation (OSDI'06)*, pages 103–116. USENIX Association, 2006.

[80] Tianyi Yu, Qingyuan Liu, Dong Du, Yubin Xia, Binyu Zang, Ziqian Lu, Pingchao Yang, Chenggang Qin, and Haibo Chen. Characterizing serverless platforms with Serverlessbench. In *Proc. 11th ACM Symp. on Cloud Computing (SOCC'20)*, page 30–44. Association for Computing Machinery, 2020.

Appendix

We discuss two additional results in the Appendix: (1) memory usage analysis of different runtimes on LangBench, and (2) other speedups from the runtimes over GCC.

Resource Usage: Memory

Figure 7 shows the peak memory usage of the different runtimes. Compared to Figure 2, it also shows the completion time under the minimum memory usage configuration (e.g., the heap size setting in OpenJDK) of each benchmark. Recall that, for OpenJDK and V8, the minimum amount of memory was set by determining the first heap configuration that did not cause a crash; for Go, GOGC was set to 5%. We then

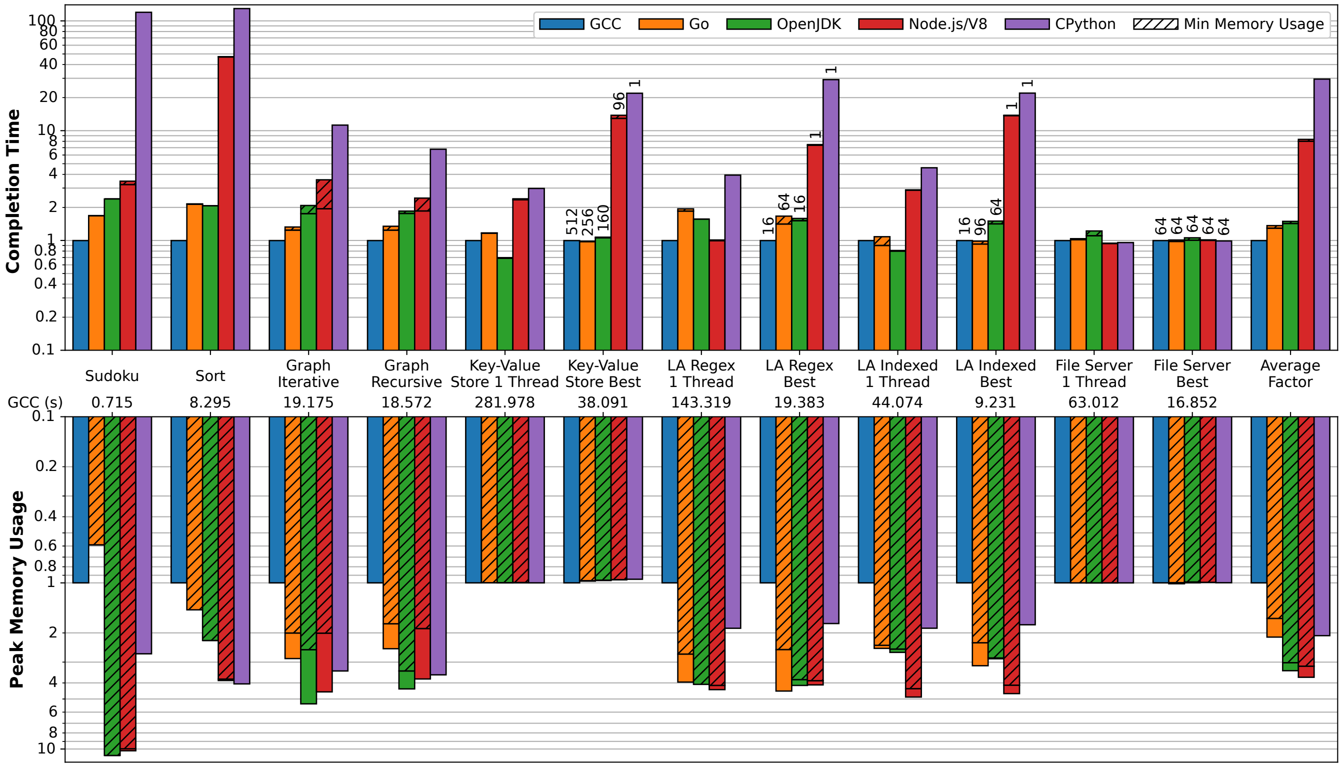


Figure 7: Relative completion time and peak memory usage for various language implementations as a multiplicative factor compared to optimized code under GCC. Each benchmark uses the most optimized version for that language implementation.

continuously increased the heap settings until performance no longer improved. Peak memory was measured using the reported maximum RSS from `getrusage`.

The figure shows that all language implementations use at least 2x more memory than GCC. The more complex runtimes are the worst offenders with V8/Node.js and OpenJDK using 3.70x and 3.38x more memory than GCC on average. Go and CPython use less memory, but still use 2.12x and 2.08x more memory than GCC on average. (As an exception, Go surprisingly manages to use 0.59x less memory than our idiomatic C++ version of the sudoku benchmark.) It is crucial for these runtimes to trade off increased memory usage and performance. Optimal performance can require increased memory usage, which prevents jobs from being scheduled when datacenters allocate resources to fit peak usage. This additional memory is also rarely returned to the OS causing reduced memory utilization.

For both OpenJDK and V8/Node.js, the two runtimes that require the most memory to achieve optimal performance, their worst case was the sudoku benchmark with OpenJDK using 10.94x more memory than GCC. The sort benchmark has the lowest memory usage with GCC only requiring 3.42MB. However, OpenJDK and V8/Node.js also had benchmarks that did not have any memory overhead when compared to GCC. Both runtimes used the same amount of memory as GCC for the key-value store benchmark, despite it being the next smallest benchmark with GCC requiring only 33.43MB.

CPython's peak memory usage was the closest to that of GCC, but requiring 2.12x more memory on average. It also had the lowest worse case, requiring 4.06x more than GCC for the sort benchmark. Despite being more memory efficient than the other runtimes in most of the benchmarks, CPython still used more memory than any other runtimes in the sort benchmark. Go was also able to use less memory than CPython for the sudoku and graph colouring benchmarks.

In fact, Go was the only language implementation able to use noticeably less memory than GCC, using 0.59x less in the sudoku benchmark. Upon inspection this stemmed from our version of the benchmark using the C++ standard library. The complete C++ version peaks at 3.46MB of RSS, but almost all of this memory is allocated immediately upon running the program. We found that using a C++ implementation that did not use `iostreams` but instead used `open` and `read` reduced the memory usage to 2.72MB. However, the largest improvement was from not linking the C++ standard library by removing the `-lstdc++` flag when compiling. This dropped the usage to 1.33MB and well under Go's 2.05MB.

Runtime Speedup from Library Implementations

In addition to the cases discussed in §8, there are other cases where managed runtimes performed better than GCC. Go performed better than GCC on the indexed search log analysis benchmarks, taking 0.90x and 0.93x less time for the single and multithreaded versions, respectively. Further, V8 has the same performance as GCC on the single-threaded regular expression based log analysis. In both cases, the good performance comes from the library implementation of pointer copying (in the case of Go on indexed log analysis) and the regular expression engine (in the case of V8 on regular expression based log search).

Surprisingly, whereas Go spends a total of 0.04 seconds in a critical section in indexed log search, GCC takes 2.49 seconds. In the log analysis benchmarks, each worker thread returns a list of matched log messages to the main thread by appending a thread local list of results to the main thread's global list, while holding a lock. Inside this critical section, for the append operation, Go copies more pointers per loop iteration than GCC.

In Go, appending to a list (referred to as a slice) is done efficiently because slices are an intrinsic type and appending is performed by a builtin function, which uses hand written assembly. Both GCC and Go use 128-bit wide XMM registers [9] to move two 64-bit pointers at a time with a single `mov` instruction. The assembly in Go unrolls the loop as much as possible, using all 16 XMM registers to move 32 64-bit pointers per iteration. Furthermore, rather than checking if a write barrier is required for each pointer, Go checks once if write barriers are required for all pointers, as they do not need to be performed when concurrent marking is not active [8].

For the same operation in C++ with `std::vector`, GCC only moves 2 64-bit pointers in a single XMM register per iteration. Therefore, for every 2 pointers (or single XMM register move) GCC must also execute a compare and jump instruction to iterate the loop. On the other hand, Go will only execute these two loop iteration instructions every 32 pointers (16 XMM register moves). Furthermore, because we use `std::unique_ptr`, GCC must set the pointers in the thread local vector to `NULL`, as ownership has been transferred to the global vector. GCC stores `NULL` to 2 pointers each iteration of the loop using another XMM register.

[8] Write barriers are explained in more detail in Section 6.3.