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

There wasn't any research that compares the performance in-depth of popular managed languages, JS, Python and Java, to each other or to compiled languages like C++ and Go. We built benchmarks and instrumented the runtimes for JS, Java, and Python, and determined which is faster (under our benchmark) as well as why performing GC in Java can result in net speedup when compared with C++.

The most widely used programming languages today are managed languages. The three most popular languages on GitHub since 2015 are JavaScript, Java, and Python [12]. These languages offer the promise of productivity and thus faster product-to-market 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.

When selecting a programming language for a new service, the performance of the language is rarely a consideration at the outset, as engineers frequently opt for productivity, 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"* [20].

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. 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."* [16]. Performance issues are also cited as the main reason Twitter was forced to switch from Ruby on Rails to Scala and Java [14, 15].

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++ [9], others suggest that Java programs can be faster than C, because the JIT compiler produces faster machine code by leveraging a runtime profile [8]. Similarly, there have been polarized debates with respect to the performance of JavaScript [3, 13], Go [16, 19] 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].

Discussions on *scalability* are even muddier. For example, in a popular blog by the official Node.js 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 executes on a single thread [18]. 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 serializes 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). 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. But 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 [17], V8/Node.js does not expose any performance counters to report this overhead.

In this work, 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 [5,7,10,12]. 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. We chose these languages not just because of their popularity, but also 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.
- ***Execution modes.*** Only Go is ahead-of-time compiled. OpenJDK and V8/Node.js first interpret bytecode, and compile hot functions Just-In-Time (JIT). CPython only has an interpreter and no JIT compiler.
- ***Concurrency models.*** V8/Node.js is event driven where event handlers are executed sequentially on a single thread. Similarly, CPython's GIL only allows one thread to execute 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.

1 Instrumentations and Benchmark Suite

We instrumented three runtimes: OpenJDK, 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 [26], the paradigm shift to the cloud [23,24,27] and data analytics [22] 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 [23,24]; the problem is also exemplified by short-running instances in Function-as-a-Service platforms [23,27]. In 2020, the median AWS Lambda invocation ran for only *60 milliseconds* [11], while startup times for the JVM and V8 are on the order of hundreds of milliseconds or even seconds [22, 23, 27, 28]. Similarly, data analytics systems face a fundamental tension between parallelizing long running jobs into shorter tasks and the runtime's start-up overhead [22]. And bytecode-level profiling can enable effectual optimizations. For instance, Instagram engineers instrumented CPython to identify the bytecode instructions with high overheads, and then optimized their code to avoid using these expensive instructions [6].

Why profile type and bounds checking? Dynamic type and bounds checking is a major source of V8's overhead, as we will show later. Similar to bytecode profiling, programmers can optimize their JS programs to avoid such overhead once the source is identified. Our instrumentation also enables eliminating type and bounds checking 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]`. 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 for any 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).

Benchmark suite. We created 6 realistic applications from the ground up. The applications, which range from microbenchmarks 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. Three of the six applications are parallel, and we parallelize them using both multithreading and multiprocessing where applicable. From the six applications, we created twelve benchmarks by varying degrees of concurrency, and exploring alternative implementations of the applications. We implemented these applications in each of the 5 languages.

While the *design* of every implementation of an application is conceptually identical, each *implementation* is optimized for the language: we used our best effort to make the code idiomatic. The benchmark suite is called ***LangBench***. The source code of our instrumented runtimes and LangBench can be found at `https://github.com/topics/langbench`. More details can be found in our ATC paper [21].

2 Overview of Results

Figure 1 shows the run times for the benchmarks in LangBench. Unsurprisingly, 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 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.

Next, we use our instrumented runtimes to provide detailed analyses that explain these results.

3 Runtime Overhead (Single-thread)

This section investigates the source of runtime overheads on single-threaded applications that performed poorly. Specifically, we found (1) type and bounds checking (§3.1) is the bottleneck for V8 in its slowest benchmarks (Sudoku and Sort); (2) interpreter performance (§3.2) 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 (§3.3) can be the bottleneck for both OpenJDK and Go, even when heap usage is small.

3.1 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 Sudoku and Sort benchmarks, which are the two single-threaded

	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 1: 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 2. We measured the resulting execution time, and compare it against the default execution time with all checks.

benchmarks where V8 showed the worst performance compared to GCC. Next we zoom into the Sudoku benchmark to explain this overhead.

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 2. Each dimension requires 5 checks, and the 11th check is used for the final value. Table 1 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 an integer. Second, V8 must similarly check that `x` is an integer, rather than an object. Omitting these checks made it 8.1% faster (Table 1) — 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. 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.

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. 1). 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

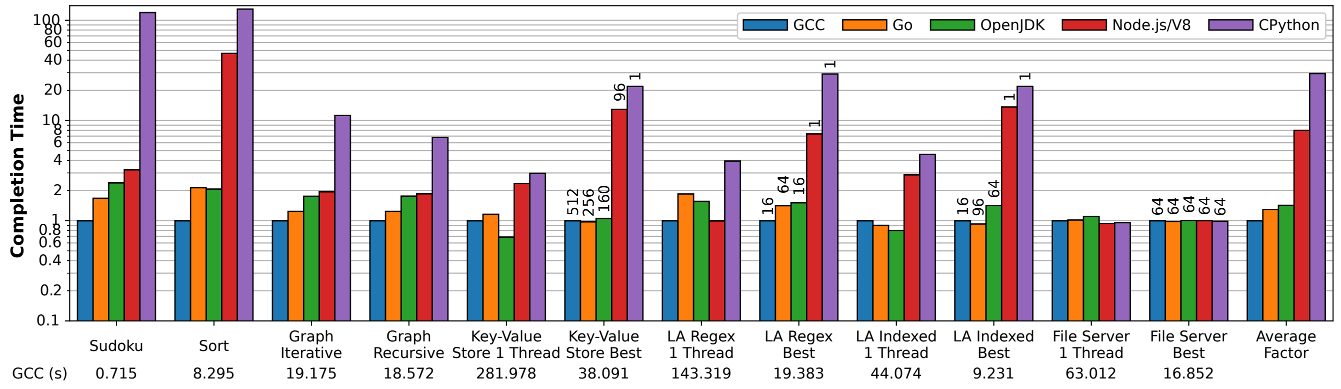

Figure 1: Relative completion times for various language implementations normalized to GCC. Note the logarithmic scale of the y axis. Sudoku, Sort, etc., are different benchmarks under LangBench ("LA" refers to the log analysis benchmark). The numbers at the bottom shows the benchmark's absolute execution 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 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 §4). As for the file server benchmark, both Node.js and CPython's best performance is achieved when using 64 server-side threads (§4). The number of server-side threads in Go is automatically determined by the runtime as described in §5.2. The number of threads for log analysis is the number of worker threads (as there is no client).

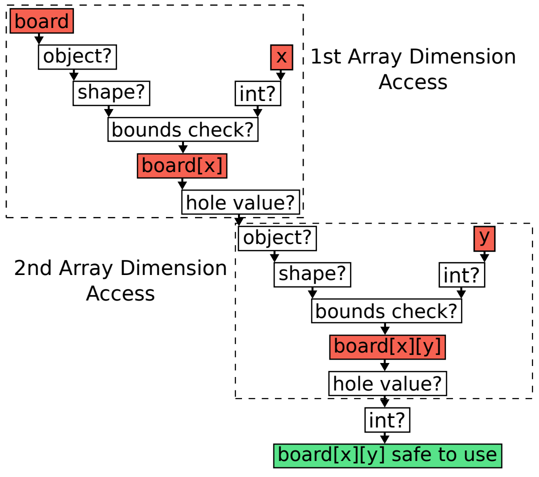

Figure 2: Checks required to access `board[x][y]` in V8/Node.js. The meaning of each check is explained in §3.1.

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.

3.2 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 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 2 shows the performance profiling results of different bytecode executions, using our instrumentations.

Bytecode		Insn. per BC	Cycles per BC
OpenJDK	`aaload`	12	7.7
	`iaload`	11	7.1
Node.js	`LdaKeyedProperty`	90	26.3
CPython	`BINARY_SUBSCR`	138	41.8

Table 2: Statistics for array access bytecodes (BC) performed by various interpreters for the sudoku benchmark.

CPython is still 2.07x slower than V8, even though both of them do dynamic typechecking. As shown in Table 2, 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 V8's intermediate representation (IR), whereas CPython is implemented in C. Note that OpenJDK's interpreter is entirely in hand-crafted x86 assembly.

3.3 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. 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 [4], adds 44 instructions for these write barriers, completely dwarfing the 6 instructions required to swap the elements and 5 for bounds checking.

4 Scalability Limitations

We found that CPython and Node.js limit the degree of parallelism achievable. 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. 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 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

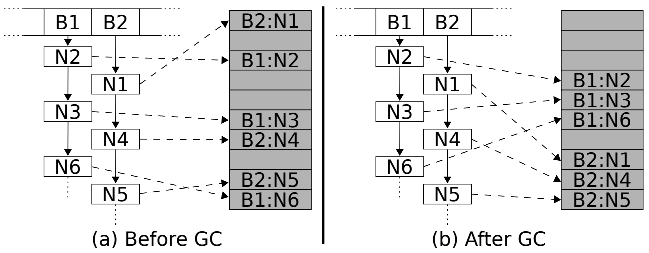

Figure 3: The key-value store before and after a GC pause. White boxes logically represent Java objects, and the shaded boxes represent the objects' location in the JVM heap. A 'B' denotes a bucket mapped to by the hash function, and an 'N' denotes a node in the bucket's linked list. The number of the 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.

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.

5 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 [25]. 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.

5.1 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 collisions are added to a bucket by appending the key-value

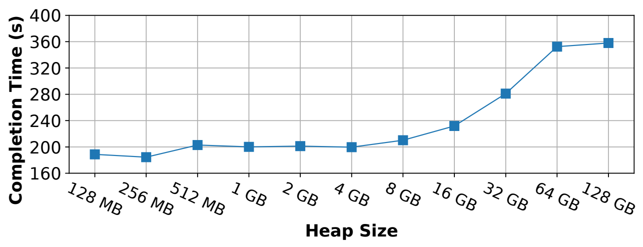

Figure 4: The OpenJDK single threaded key-value store benchmark run with increasing heap sizes, corresponding to fewer GC cycles.

(KV) pair to a list. This is shown as the white boxes in Figure 3. 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 3 (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 3 (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 4 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.

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. 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.

5.2 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.

5.3 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. 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.

References

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

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

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

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

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

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

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

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

[9] 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.

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

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

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

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

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

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

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

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

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

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

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

[21] David Lion, Adrian Chiu, Michael Stumm, and Ding 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 *Proc. USENIX Annual Technical Conf. (USENIX-ATC'22)*. USENIX Association, 2022.

[22] 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.

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

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

[25] 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.

[26] 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/.

[27] 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.

[28] 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.