ABSTRACT
There exist many preconceived notions about how C, Java, and Go perform in various different applications. Our research attempts to uncover and quantify how the programming paradigms, data models, compiler optimizations, and other artifacts of various programming languages affect performance in different application domains.

By applying these different languages to a selection of algorithms and benchmarking those algorithms with different compilers and optimizations, we develop a better understanding of how different languages perform and are able to better quantify their tradeoffs. Specifically, our research produces data on the tradeoffs between speed and memory efficiency of these languages, as well as documenting the process of optimizing and refining our code to reach our presented results.

1. INTRODUCTION
In order to overcome the excess of anecdotal claims against and in favor of programming languages, there is a clear need for objective data relating to the performance of different programming languages in various application environments.

1.1 Motivation
The problem that exists today is that there is little solid data addressing how different languages and programming paradigms affect speed and efficiency and how exactly that performance trade-off should be evaluated. Despite this, there exists a tremendous stigma in favor or against various languages, without solid evidence to support it. Specifically, as explained above, one common perception among programmers is that C is faster than Java due to its lack of memory safety and garbage collection. Perceptions like these shape the decisions made by software engineers in real production environments.

Our research produces controlled data to quantify and explain the tradeoffs that arise from the use of different languages and programming paradigms. Our case studies compare a number of metrics measuring the speed and memory-efficiency of several languages by exploring how changing language constructs and compilers can affect these metrics. This allows for the results of our research to provide a basis for language choice in various real-world fields and applications.

1.2 Overview
Our research specifically addresses the performance of the languages C, Java, and Go. Our benchmarks produce reproducible results quantifying the performance tradeoffs of using these programming languages and using different language constructs within each of these languages. Various metrics are repeatedly measured and tweaked to further optimize each algorithm implementation in each language; included in these are features such as language constructs, data models, compilers, and compiler optimizations.

By implementing each algorithm multiple times utilizing different constructs within each language, while also varying compilers and compiler optimizations, our research controls for programmer ability and produces data that measures the performance of these languages and algorithms. In addition, while improving the implementations in each language, the programming style was intentionally matched across languages to compare the relative effects of the optimizations in each of the languages. These precautionary measures assure that programmer ability and style have a minimal effect on the variation and results of our data.

2. BACKGROUND
The key metrics relating to resources required to run a program consist of the speed required to run on an isolated machine and the memory efficiency of the program.

• Speed: The speed of a program refers to its average CPU time runtime on a particular system. The speed of a program is a function of the language and constructs chosen to implement it, as well as the system it is run on.

• Memory Efficiency: The memory efficiency of a program refers to the amount of memory required by the process running the program. Similar to the speed of a
program, efficiency can also be greatly affected by factors like the language, data models, and algorithmic complexity used to implement the program.

In general, certain languages have the stigma of being fast because they sacrifice certain security checks, for example checking the safety of accessing a memory location. Providing memory safety in a language would seem to require overhead computations that would slow a program down, leading to this notion that such languages are faster [9]. The memory efficiency of a program alone can have a large effect on dictating what applications an implementation may be well-suited for due to its memory requirements. This can be dictated by the data structures used to model the data required for the algorithm.

These two factors can also have some amount of dependent interaction depending on the language. Memory management in low level languages is either performed automatically by a garbage collector or it is managed manually by the programmer. Go and Java both take care of allocating and deallocating memory automatically; however, C requires manually allocating and freeing any memory that is allocated on the heap. This finer control over exactly when memory can and should be freed may be beneficial in applications with large memory usage where memory is limited.

There exists a stigma that garbage-collected languages must be slow due to the interaction with the garbage collector running during the program execution, which is not necessarily true [4]. In addition to the performance of the program, both speed and memory efficiency can also be affected by how easy the language is to program in. In some applications, a small sacrifice in program performance might still outweigh a large increase in programming-ease and speed of development.

Ideally every program would be both fast and secure; however, this is a difficult pairing to achieve with the existing languages and tools at hand for programmers today. Languages like Java are thought to be more secure, at least with regards to memory allocation, because they have memory bound checking [9]; however, this does not necessarily mean that the Java Virtual Machine (JVM) itself is safe from memory vulnerabilities. Java has a stigma associated that it cannot be as efficient as more memory-vulnerable languages like C [12]. Conversely, supposed efficient languages are vulnerable to significant security threats, like the common buffer overflow attack.

Another factor that can significantly impact the speed of a program is the compiler. Compilers convert human readable code into machine code. This is an important step for all performant low-level languages because it allows for many code optimizations to take place. Compiler optimizations can make a program faster or slower, or one can even use certain “safe” compilers that make the program safer with respect to security, at the cost of speed. Programmers will always be faced with the challenge of balancing the speed and security requirements of their application, making the quantification of the trade-offs between languages and compilers extremely important.

3. RELATED WORK

Existing research has touched on the performance trade-offs between different languages. Below are some examples of existing work performed in this domain and how their methodologies will be incorporated or improved upon to fit the needs of our data and research. Specifically, we will talk about implementation details (Sec. 3.1), language constructs (Sec. 3.2), machines and compilers (Sec. 3.3), and prior research performed on specific relevant languages (Sec. 3.4).

3.1 Implementation Details in Benchmarking

The Computer Language Benchmark Game [3] has done a significant amount of work analyzing the speed differences between programming languages. By running different implementations of the same algorithm submitted by various participants online, they claim they are able to control for the code style of their implementations. Other research studies have also made attempts to control for variability in implementation choices by having many programmers implement the same program for their experiment [14].

While these strategies are fairly effective, they are only able to capture general trends in performance between languages. By grouping all implementations for each language together, they are able to mitigate the effect of code style by ignoring it altogether and generating aggregate data for the numerous implementations. Our research, instead, attempts to avoid stylistic effects on performance through style-agnostic algorithm selection, explicit style-matching across languages, review, and proactive measurement of possible performance factors such as varied language constructs or style.

3.2 Language Constructs in Benchmarking

Although empirical analyses of programming languages have been performed in the past with the intention of generating hard data [13], as our work intends to do, these studies fail to capture the performance impacts of different language constructs. These studies typically consider only the implementation time of the programmer, memory usage, and program structure. Our research instead measures the performance impact of certain language constructs within each language and across languages to give a fuller, style-independent representation of the performance in each language.

Speed and efficiency can often be affected greatly by the language constructs being used. The Object-Oriented (OO) nature of the Java language, for example, can make even simple tasks over-complicated, forcing the Java runtime to call hundreds of methods and create large numbers of unnecessary objects in the process. Mitchell, et al. [12] discusses how Java overhead memory requirements are often found to be up to 60-80% of the total required memory of a program in deployed applications.

Comparisons between Java and Scala have also conclusively shown that the choice of data structure and usage in the implementation of an algorithm is extremely important and can greatly impact the performance. With respect to object-oriented languages like these, performance can vary largely by cleverly using primitive types over objects [8]. Similar comparisons of the performance of shared language constructs such as do-loops, for-loops, and array-accesses have also been performed [2]. This data is useful to mitigate stylistic performance effects and to see the granular performance impacts of these varying constructs in each language.

Our research is utilizing the results of previous studies to carefully identify key performance-affected characteristics of Java, C, and Go and vary them in our implementations.
This generates measurable data demonstrating the costs of specific constructs, for example object-oriented paradigms, and how these measure up against the constructs in other languages such as structs in C or Go. Our research builds upon other attempts at profiling the performance of languages such as Go using different constructs of the language and measuring the performance of a program after small modifications [6]. By performing these granular comparisons within and between each language, we can determine if there exist advantages that certain languages may uniquely possess for certain applications [2].

3.3 Machines & Compilers in Benchmarking

Programming language benchmarks can be extremely enlightening by just showing the vast performance differences possible within one language. Research into the efficiency of Java, for example, has illuminated how core characteristics about the language and its compiler can massively affect its performance. Different interpreters and compilers have been used and compared against equivalent C programs [11]. These performance differences can be affected by data structure choice, as mentioned above, or by other factors such as the compiler [7] or machine running the program [3]. One such study measures the memory usage for several tasks, as well as the CPU seconds to perform various other microtasks [13]. Our research similarly measures memory and CPU metrics of each of our algorithms on an isolated machine, while also varying different parameters such as compiler and implementation.

3.4 Specific Language Comparisons

Some research has been performed benchmarking C, Java, Scala, and Go on a loop-unrolling algorithm, building the implementations to be as similar as possible across languages. In addition, fully-optimized versions of the Java, Scala, and Go programs were also included in the benchmark to see how performance can vary between and within different languages. The measured benchmarks include code size, compile times, binary sizes, memory footprint, and runtime [5].

Our research is utilizing similar tweaking of algorithms to quantify the different tradeoffs between these languages when written with different implementations. Our research extends on many of these prior efforts by considering several different algorithms with some of these languages.

Additional research has compared performance between languages in highly specific domains. For example, one such study compared the speed and efficiency of Java, C, and C for interactive scientific visualization and graphics rendering [15]. Another study looked at Java, C, and C# qualitatively for the specific application of developing distributed systems in a web environment [10].

In addition to formal research, there exists a wealth of online blog posts and discussion on the issue. Although programmers discuss the issue at large, the data presented in these forums is highly uncontrolled and informal. Thus our research formally attempts to control for any factors which may affect our measurements, producing concrete performance data that can be supported and reproduced.

4. SYSTEM MODEL

Our research requires a systematic approach to how we generate, collect, and analyze our data. We can break down our workflow into a few main components: build benchmarking suite, write each algorithm in Go, Java, and C, refine the algorithm implementations by writing code that tests different features of the language and uses different programming paradigms, compile all of the code with all of the different compilers we are testing while varying optimization flags, and perform a statistical analysis of our data. This is laid out in our block diagram below, See Figure 1.

![Figure 1: Research Workflow](image)

Our benchmarking suite is used to record the efficiency of the program. More specifically, the benchmarking suite measures the runtime and memory usage of the program.

Once we verified that our benchmark suite produced accurate results, we moved on to writing code. We have written the following algorithms in Go, Java, and C.

- **NBody Force Calculation** - Calculates the force, acceleration, velocity, and position of N bodies each influenced by the gravitational force exerted on each other. We used five bodies modeled after a subset of bodies in our solar system.

- **Levenshtein Edit Distance** - Recursive, dynamic programming algorithm that finds the number of edits required to change one string into another. For example, Levenshtein(cat,bat) = 1, and Levenshtein(dog, real) = 4.

We chose these two algorithms because they test different things about a language. Mathematical algorithms such as NBody test how good the internal mathematics functions of the language are. Algorithms like NBody with large data structures test how languages and paradigm shifts with regard to dealing with large data structures in memory. Levenshtein Edit Distance tests how languages deal with a quickly growing stack, tail call optimizations, calling conventions, and recursion depth.

Once we wrote these algorithms, we ensured that they were correct and that we had to the furthest extent possible removed the individual programming style. This was done by peer review style checks as well as refinements to the algorithms as a group. Once we knew that the programs were correct, we began to vary and optimize the implementations. Varying the implementation allowed us to then test different language constructs. We only varied the code at the data model level, because changing the implementation...
at the time complexity level would result in a speedup that is only an effect of a faster general algorithm.

After we wrote all of the code, we compiled our code with different compilers and compiler flags. The compilers and optimizations we looked at are:

- **C**
  - gcc (-O0, -O1, -O2, -O3)
  - llvm (Clang) (-O0, -O1, -O2, -O3)
- **Java**
  - javac
- **Go**
  - gc (Go Compiler)

After we generated our binaries, we ran our benchmarking suite on the programs and recorded the data. This collection of data is the crux of our research, so we were sure to use multiple trials and test our benchmarking suite extensively to ensure consistent results.

Finally, we performed analysis on our data to determine the most performant language for each scenario. The term “most performant” refers to a combination of speed and memory efficiency. The most performant language or implementation in a scenario depends on the goals and resources available, but generally would be the fastest running and most memory efficient implementation of a language.

5. **SYSTEM IMPLEMENTATION**

In order to achieve the systematic and controlled experimentation environment we seek for our benchmarks, a consistent procedure was essential for measuring algorithm performance. The core of our system implementation involves implementing our selected algorithms and the benchmarking tools used to measure runtime and efficiency performance metrics.

5.1 **Algorithm Implementation**

5.1.1 **Levenshtein Edit Distance**

We developed two implementations of the Levenshtein algorithm. In all languages, the first implementation was written naively by using language constructs to take a substring/subarray in that language. In Java this meant calling `String.substring` to allocate a new string, in C, using `memcpy` to copy a substring, and in Go using slicing to change the referenced region of a slice, which does not allocate or copy any memory. The algorithm was then optimized to pass only indices and the original strings to represent substrings, avoiding copying memory altogether.

After observing the results of these implementations, the research was further expanded to emulate a more realistic production environment. Instead of naively calculating the various values possible in the space of possible answers, the algorithms were “memoized” to avoid repeated calculations of the same function with the same inputs. Effectively, the intermediate results were cached within the program. In memoizing the Levenshtein algorithm, both recursive and non-recursive versions of the program were implemented in each language; however, the performance increase of the memoized implementations was so significant that they were not relevant to our analysis.

5.1.2 **NBody**

We developed multiple implementations of the NBody algorithm in each language to model the various ways the data can be laid out and accessed. In each language this means utilizing object-oriented class design to create objects or structs versus storing the data in a structure of parallel arrays. In C, the structs can also be explicitly allocated on the heap or the stack.

![Visual representation of an Array of Pointers data layout](image)

![Visual representation of an Array of Primitives data layout](image)

In Java, one implementation used an array of Body objects that held data including mass, velocity, and position. The second implementation had multiple arrays of primitives, forming a structure of arrays rather than an array of Body structures (objects). The program thus constructed parallel arrays for mass, velocity, acceleration, etc. These two data model layouts are visually represented by Figures 2 and 3, respectively. The reasoning for this is when creating an array of Body objects, “new” must be called each time. By putting just the primitive values we care about into separate arrays, there are no more objects or “new” calls. Another optimization was inlining functions into the body of the main function to essentially remove all function calls.

C and Go both are able to represent data in three distinct ways. In addition to Java’s array-of-pointers and array-of-primitives layouts, C and Go offer a array-of-structs data model which consists of a contiguous block of allocated memory. This data model is visually represented in Figure 4.
In C, the implementations varied on what was passed to the methods that calculate the force changes. The first passed through the entire struct, the second passed a pointer to the method and returns the modified struct, while the third passes a pointer to the struct and modifies the struct in place without returning anything. Another implementation used malloc and free to allocate memory for the structs. As with Java, there was also an implementation where all the functions were inlined.

In Go, the algorithm implementations were similar to those for C due to the similar nature of the data models. There were implementations that used arrays of structs and arrays of primitives. Like C, implementations varied the calling style to use pointers to structs as opposed to the struct itself. This also allowed for optimizations like modifying a struct passed by reference in a function that returns void. This prevents extra assembly calls which may slow down performance.

5.2 Benchmark Implementation

The implementation of our benchmark focuses on speed and efficiency, which are two essential metrics to software performance. These metrics highlight the supposed shortcomings and differences between how these languages interact with and run code; for example, memory usage and runtime benchmarking show us exactly how our program is executing in Java, where the mark and sweep garbage collection is speculated to have significant performance impact. This also hopefully sheds some light on how good Go’s garbage collector compares to manual memory management in C.

In order to accurately and consistently measure the runtime of a program, our benchmark utilizes the Unix `time` command to measure the time it takes for the program to be executed. This ensures a consistent timing mechanism across the varying languages, as opposed to the option of using language-specific timing mechanisms within our implementations. By making the programs have runtimes on the order of 5-10 seconds, we can get more accurate readings than trying to differentiate between runtimes that are all under a second. Longer runtimes mitigate the program startup times which can be inconsistent between languages, especially with the Java runtime startup time. This implementation choice also mitigates the potential effect of timing benchmark method calls affecting the performance of the algorithms being measured.

Memory usage of the programs is also measured by the Unix `time` command. The exact memory statistic that we use is the maximum resident set size of the process during its lifetime. Because the algorithms we are looking do not constantly allocate and free data, this metric is useful to show how different data representation can affect memory footprint in different languages.

Finally, our research requires the use of a dedicated machine or cluster of machines. A dedicated machine will run into minimal issues regarding other processes running concurrently or spikes of CPU activity that could impact our data. While some noise will exist, it is faint enough to be ignored. We acquired access to a dedicated cluster of 128 64bit dual-socket dual-core Core 2 Duo Xeon processors running at 2.66Ghz with 4mb of second level cache. [1]

6. RESULTS

The results of this research are the runtime and memory use data across the different languages and implementations of the previously mentioned algorithms. This data is summarized here.

Figure 5 shows the relative runtimes of the fastest and slowest implementations for the NBody simulation with the speedups relative to the fastest C implementation. The fastest C implementation outperformed all other implementations. Java’s fastest was three times slower as C’s fastest implementation and had about the same performance as the slowest implementation in C. The Go implementation was slightly slower than Java but comparable. However, while the slowest Java implementation was on par with the slowest C implementation, the slowest Go implementation was almost 13x slower.

6.1 NBody Results

![Figure 5: NBody performance across programming languages.](image)

Figures 6 and 7 outline the differences between individual implementations in C. The results are broken down by the data model, compiler, and optimization flags used for that corresponding data. They are normalized to `Structs Clang -O1`, the fastest implementation. The `Structs` implementation used stack allocated structs. `Primitives` used arrays...
of primitives. Heap used heap allocated structs. Looking at each implementation across a single compiler optimization (O0), the stack allocated structs were the slowest, followed by heap structs, then by the primitives arrays. The Primitives implementation was 1.5x slower than the fastest implementation and Structs was almost 2.0x slower. However, as more complex compiler optimizations were used, the runtimes leveled out. Regarding memory use, it stayed relatively constant with the Heap implementation using slightly more.

Figure 6: Runtime and memory usage of C implementation of NBody

Figure 7: Runtime and memory usage of C implementation of NBody

Figure 8: Runtime and memory usage of Go implementation of NBody

Figure 9: Runtime and memory usage of Go implementation of NBody

Figure 10: Runtime and memory usage of Java implementation of NBody

6.2 Levenshtein Results

As seen in Figure 10, the results for the Java implementations were almost identical. The Performance Optimization implementation, which was fully inlined and used structured arrays of primitives, was the fastest, but only slightly and not significant.
crease can be attributed to the reduction of method calls in favor of while-loop iteration.

![Graph: Slow Levenshtein Runtime w/ Compiler Optimizations](image)

**Figure 11:** Comparison of runtime of different C compilers on Levenshtein implementation

Lastly, Figure 12 compares the fastest, non-memoized implementations for Levenshtein Edit Distance, normalized to Java. C and Go were 1.6x slower than Java, while having roughly equal performance with respect to each other. This implementation had no optimizations, and Java outperformed both C and Go, albeit slightly. The performance discrepancy between Java and the other languages is unclear to us even after extensive exploration of the potential root causes in the implementations. Thus, we attribute the performance increase to be related to optimizations the JVM and JIT compiler is able to make to optimize the call stack when performing this deep recursion.

![Graph: Language Comparison of Non-Memoized Levenshtein Edit Distance](image)

**Figure 12:** Comparison of fastest Levenshtein implementations between languages

7. CONCLUSIONS

Rather than make a single strong conclusion about which language is the fastest or most performant, our conclusions about the languages draw from the optimization process and how performance changed over the different implementations. Ultimately, there is no all-purpose “best” language, but rather a series of trade-offs that must be faced depending on the requirements and domain of the application.

7.1 The Optimizations

Over the course of optimizing our implementations to test different pieces of each language, we uncovered the driving factors behind algorithm performance and performance deviations in our data. Overall, the implementation of the algorithm can matter significantly.

While optimizing the C implementation of NBody, we found using pointers and primitives is much more performant than passing objects around. In C, the transition from passing through objects to helper methods, to passing pointers and modifying the objects in memory increased performance. This was because less things were pushed onto the stack and accessing memory via pointers is much faster than passing redundant data. Also in C, we removed the `Math.pow` call which reduced the runtime significantly, as well as decreasing reliance on external libraries. This external library call initially made the C implementation slower than Java; once the function call was removed, the C implementation became faster than the Java implementation.

The process of optimizing the Java implementation of NBody began with a naive implementation taking full advantage of Java’s object-oriented language constructs. By changing the implementation to utilize a structure of parallel arrays of primitives rather than another Java class, *Body*, the performance of the algorithm improved slightly. The new implementation actually required slightly more memory than the initial implementation; however, by performing the optimization of eliminating object creation and the reducing method calls in favor of accessing arrays of primitives and inlining computation provided a slight improvement in the performance of the algorithm.

After further inspecting the implementation, another optimization was made to eliminate two seemingly unnecessary arrays altogether in favor of a local variable and an inlined computation. This optimization, surprisingly, slowed the runtime of the algorithm slightly, causing a 4% slowdown from the previous implementation. Looking closely at the performance of this implementation, it became clear that the function inlining had a positive effect on performance, but utilizing a local variable rather than manipulating values in an allocated array caused a slight performance drop, making the overall performance much worse. This decrease in performance can be attributed to Java having to allocate and garbage collect memory for the local variable repeatedly, in fact $O(n^2)$ times in this algorithm. In order to overcome this performance hurdle, the implementation was refactored again to revert the use of the local variable in favor of an array, once more. This final implementation ultimately yielded the best performance, as it only had to manipulate and perform arithmetic on the existing values in the array rather than repeatedly allocating and deallocating memory for new `int` objects. The performance increase of this change yielded a 1.06x performance speedup compared to the previous, slow implementation and a 1.04x performance speedup compared to the original object-oriented implementation.

Both Java and C implementations were ultimately entirely inlined, removing all function calls, which increased performance in both languages. The decreased reliance on passing objects and helper methods in both C and Java meant better performance, however the advantage of having direct access to memory pointers in C yielded better performance than Java.

Optimizing the NBody simulation in Go brought on many questions regarding garbage collection and the use of point-
Java bytecode. Then, the JVM interprets the Java code and runs it on the processor. While this is beneficial for portability and security, it requires significantly more resources for the JVM to run. The large memory overhead for Java is due to that JVM and is unavoidable, regardless of how optimized the program is.

Since our implementations did not manually allocate or deallocate memory, the memory used is the memory used to store the data we are working with. In Java, the space taken up by the objects we allocate is higher than in C. In Java, the memory could be allocated anywhere and is handled by the virtual pages in the JVM. In C, the structs are just contiguous bytes. Go is laid out similarly to C and follows the same reasoning.

Like C, Go allows for pointer types to reduce the overhead of passing large structs into functions. Unlike C, Go’s garbage collector creates extra overhead when using pointers. Using pointers in Go served little use. The memory indirection cost from dereferencing the pointers was large. In addition, the NBody body structs consisted of 40 bytes, which means that you need 48 bytes to hold the data and the pointer to it. The vector struct was only 16 bytes which meant that using it as a pointer brings the total to 24 bytes. This overhead may make sense when optimizing a program that uses structs that are on the magnitude of hundreds of bytes, but in our case, that overhead was immense and drastically affected both runtime and memory usage. Go’s memory model abstracts away ideas like the stack and the heap which makes programming easier. Unfortunately, this means that as a programmer, you can not take advantage of spatial locality of your data if you choose to use pointers because it is unknown where the struct actually resides in memory. Furthermore, when returning pointers from functions, the garbage collector has to track when that struct can be freed, thus adding more overhead when using pointers. In the inlined versions, there wasn’t an increase in the memory usage when pointers were used which can be attributed to a lack of allocating memory in function calls.

It is worth noting that our algorithms did not test memory management so we would expect that the gap between C and Go would get larger as programmers can take advantage of smart memory management. A strong conclusion to draw from this is that because C exposes more functionality to the programmer, there is more room for optimizations that utilize spatial locality and deciding whether the data is stored on the stack or heap. Go’s memory model makes programming easier, but optimizing to the level of C is much more difficult.

8. FUTURE WORK

Future research will include more algorithm classes, more languages, deeper analysis of the assembly and generated code, and how the algorithm classes interact.

The set of algorithms as well as the set of languages that we covered, while useful, are small subsets of the algorithms and languages available to programmers. To make our data broader future research into more algorithm classes and languages such as Python, Ruby, or Scala would generate more data for a broader range of developers. In addition to new algorithm classes and languages, exploring the memory models and how they affect performance would be a priority for further research. Writing code that tests the efficiency of the garbage collection, both in terms of memory footprint as well as runtime spent performing garbage collection(mark
and sweep or reference counting). In both the new algorithm classes and languages, as well as the ones we have already looked at, deeper research can be done to examine the generated code and see how and why the assembly or bytecode is generated for each algorithm and how the speed and efficiency varies per each implementation, algorithm, and language. Picking apart the assembly will show why these optimizations, compilers, and languages make the implementations run as quickly or slowly as they do. We have already done this to an extent with the gcc -O2 and -O3 optimizations, but further examination of this information will yield more intricate conclusions.

Examining the maximum memory usages of the different implementations only tells us little about how a particular language deals with memory. Looking at memory usage over time in addition to writing programs that test a language’s ability to allocate and free memory efficiently may give insight into where certain languages excel.

9. ETHICS

The two ethical concerns raised by our research relate to the code that backs our data and how using conclusions from it could not produce our promised results.

The first ethical concern is that we are not infallible programmers and although we worked to control for style and code quality, it is possible that our coding style does not reflect the ideal or industry accepted code where conclusions from our research would be applied. Other programmers are not infallible as well. If they were to blindly take our conclusions about which language is ultimately faster in a particular situation and apply that to their project, they might not get the best results. However, part of our findings is that doing that is not the best way to choose a language for a project. Looking at the project and what types of algorithms and programming techniques will need to be used, then seeing which language, compiler, and optimizations best cater to those is a better approach. Even then we cannot provide a one-size-fits-all conclusion and do not want to mislead in that regard.

The second ethical concern is that our conclusions are not one hundred percent certain. The code we used for our benchmarks was a relatively small set compared to all the algorithms and combinations there out there. Just because our NBody implementation showed one conclusion, does not mean it applies to all NBody implementations, those similar, or similar implementations that are combined with other algorithms. The scope of our research did not allow for that and therefore the conclusions based on our code should not be thought of that way.

10. CONTRIBUTIONS

In summary, our research has provided a case analysis of the performance of C, Go, and Java programming languages in particular environments showing that language choice, data model design, and compiler choice can have serious impacts on the performance of a program.

The performance of low-level languages like C and Java seem to indeed be superior as a result of extensive compiler optimizations, with less aggressive optimization flags showing much smaller discrepancies in performance. In addition, the performance of C in NBody suggests that it’s native environment, that is not being run in a virtual machine, allow it to run comparatively faster than Java.

Another significant point in our research revealed the cost of depending on external libraries for functionality. By replacing a Math.pow call with a native multiplication, the performance in Go and C increased nearly two-fold. Thus it is important to weigh the cost of using external libraries with the complexity of the function required in the program, to avoid unnecessary use of such libraries.

In Go, the size of the allocated structs being passed to functions should be considered before deciding on passing full structs or pointers to the structs. If the size of the structs is small, the overhead of creating pointers and performing memory indirection while manipulating the pointers can have a negative impact on performance and memory usage.

Ultimately, program performance will always be impacted most by making well-informed decisions on environment, implementation, and data representation. Although language choice can affect performance, the biggest performance impact will always come from design choices made in the implementation itself.

11. REFERENCES

http://acg.cis.upenn.edu/sge.html.

[2] Algorithmic performance comparison between c, c++, java and c sharp programming languages.

http://shootout.alioth.debian.org/.


http://attractivechaos.github.com/plb/.


