Performance Analysis on Garbage Collectors in Various Programming Languages

Yunji Kim
University of Waterloo
Waterloo, ON, Canada
y23kim@uwaterloo.ca

Jun Lim
University of Waterloo
Waterloo, ON, Canada
jq3lim@uwaterloo.ca

1 Overview

Various studies have been conducted to analyze the performance of garbage collectors based on the number of objects allocated, i.e., memory usage [1]. However, little study was being done on understanding the relationship between an object's type and the performance of a garbage collector. This research attempts to examine the influence of object types on garbage collector performance across multiple programming languages. It delves into how different object types affect performance. The following sections discuss the approach adopted in this benchmarking, the testing inputs, results, how to run the project, what was learnt from the project and its limitations.

2 Approach

This section describes the design of our benchmark, tools and packages used, platform utilized and the inputs used in the testing.

2.1 Design

2.1.1 Data Structures of Object. We employ four different data structures in our benchmark:

1. **Array**: this illustrates the simplest data structure
2. **Linked-list**: as a representative of data structures with variable-sized that change during runtime
3. **Hash table**: a more complex data structure with dynamic resizing of hash table size during runtime
4. **Heap**: another complex data structure of max heap with dynamic resizing, but with a simpler mechanism than hash table

To ensure consistency of benchmark and performance comparison, we implemented each of the data structure independent from the built-in data structures, which can be found in the file of *node*, *linked_node*, *hash_table_node* and *max_heap_node*. Each data structure is designed and implemented with the same exact algorithm in every programming language to ensure consistency of program executions.

2.1.2 Target Languages. The following five programming languages illustrated in Table 1 are selected as our target of benchmarks. We employ three GC-based languages and two non-GC-based languages.

Language	Type	Characteristics
Java	GC-based	G1 (Garbage First) / CMS (Concurrent Mark-Sweep) / Parallel collector mechanisms
Golang	GC-based	Concurrent Tri-Color Mark and Sweep algorithm
Python	GC-based	Reference counting mechanism
Rust	Non-GC-based	Ownership and Borrowing system
C++	Non-GC-based	Naive (without GC) / Reference counting (Shared pointer) / Conservative GC (Boehm-Demers-Weiser GC)

Table 1. Memory Mechanisms of Programming Languages

2.2 Overall Benchmarking Flow

The following Figure 1 illustrates the benchmarking flow on a high-level for Garbage Collector (GC) based languages (Java, Golang, Python). Each language contains a `script.py` file which simulates benchmark with a set of inputs. These parameters are passed into the `gc_benchmarking` as inputs on data structure, number of elements to be allocated/deallocated, and an optional gc percentage signifying the heap size of the GC.

In the `gc_benchmarking`, the number of iterations is preset as `1000` and the benchmark starts by creating objects with count of `ELEMENT_COUNT` of the corresponding data structure. When the iterations complete, it produces a `gc.log` file containing the details of the GC event and execution details.

Upon the execution completes, `script.py` will parse and read the `gc.log` file to analyze the details in extracting performance measurement metrics for GC including:

1. GC count: total number of GC events runs throughout the execution
2. GC pause max: maximum time spent on GC pause
3. GC pause total: total time spent on GC pause
4. GC throughput: percentage of time spent not in GC collection
5. Heap usage: total amount of heap memory allocated or utilized

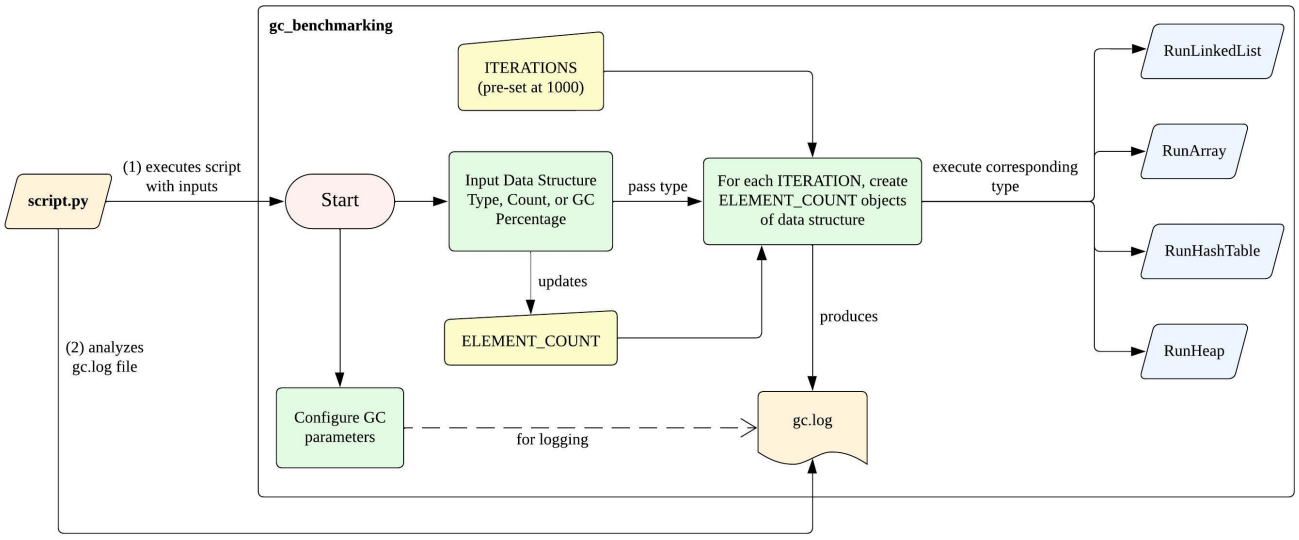

Figure 1. Benchmarking Flow

Metric	Java	Golang	Python	Rust	C++	C++ w/ RC	C++ w/ GC
Heap Allocated	✓	✓	✓	✓	✓	✓	✓
GC Throughput	✓	✓	✓				
GC Pause Max	✓	✓	*				
GC Pause Total	✓	✓	*				
GC Time Elapsed	✓	✓	✓				
# of Total GC Events	✓	✓	✓				
Total Execution Time	✓	✓	✓	✓	✓	✓	✓

Table 2. Metrics Collected Across Various Programming Languages

Here, GC pause refers to the period during the execution of an application is suspended to allow the GC to perform its collection task.

Table 2 summarized the metrics collected across all languages.

* Note that Python does not collect GC pauses metrics because as of our research, there is currently no tool/API that could extract the GC pauses details (i.e., it is done in the Python's background). Unlike Java and Golang, the GC pauses metrics are collected as the API provides for it.

For Non-GC languages (Rust and C++), since a garbage collector is not utilized in the language, no GC-related metrics are collected.

2.3 Tools

This section describes all tools involved in each programming language.

1. Java: garbagecat [2] is used to analyze the GC logs to produce a GC report that contains the GC metrics (such as throughput, pause times, gc events count)
2. Golang: No explicit tool is used, a script is wrote to parse and analyze the GC logs.
3. Python: Same as Golang.
4. Rust: We investigated three heap memory tracking tools. Our benchmarking results for Rust language are using Heaptrack.
   a. Heaptrack: Heaptrack[3] is a generic heap memory profiler that is not Rust-specific but it can be employed with Rust programs as well.
   b. Valgrind: Valgrind[5] can be used for memory debugging, memory leak detection, and profiling. It's also not specific to Rust.
   c. Jemallocator: jemallocator[4] is a memory allocator written in Rust. It is designed for performance and is used as a drop-in replacement for the default allocator ($std :: alloc :: System$) in Rust programs.
5. C++: Heaptrack and Valgrind are commonly used with C and C++ programs. Our benchmarking results for C++ language are also using Heaptrack.

2.4 Platform

The benchmarks are conducted on University of Waterloo's Linux Server [6] with the following specifications

- System Memory: 1024GB
- No. of Cores (Threads): 32 (128)
- Processor: AMD EPYC 7532 32-Core Processor
- Architecture: x86 64

3 Testing

To measure and compare the GC performance of various object types across **(1) different number of elements** , the following parameters are used as the benchmarking inputs:

1. Number of elements: 100K, 500K, 1M, 2M
2. Number of iterations (pre-set): 1000
3. GC percentages: 100%

While for the comparison across **(2) different number of GC percentages**, we use the following inputs:

1. Number of elements: 1M
2. Number of iterations (pre-set): 1000
3. GC percentages: 100%, 250%, 500%, 750%, 1000%

Notes on parameters:

- Various testings were done and due to Python's slow performance in GC, we have decreased the number of elements and the number of iterations by 10 times (i.e., 10K, 50K, 100K, 200K). We have verified that this does not impact our overall performance comparison (which will be discussed in the next section of Results).

4 Results

4.1 GC-based languages results

Figure 2, 3, 4 show the benchmarking results by running the inputs described in section 3.

The data structure type affects the GC performance. Data structures like hash tables and heaps involve table resizing or tree expansion creates more object allocations, increasing the overall time spent on garbage collection activities, thus declining the overall performance (lower overall GC throughput). On the contrary, types with fixed allocation count like arrays and linked lists have noticably better overall performance as no objects are allocated more than the specified amount.

A second set of comparison is on the various GC percentages, whose inputs as described section 3. Figure 5, 6, 7 show the results for various GC percentage. GC percentage refers to the heap size that is allocated to the GC throughout the whole program execution.

What was discovered is that higher GC percentage leads to higher heap size, which in turns reduces the number of garbage collection activites (lowering GC count, GC pauses), resulting in more time spent on non-GC events (higher GC throughput). This observation holds for all three languages, which can be observed from the figures.

4.2 Non-GC-based languages results

4.2.1 C++. C++ provides manual memory management, which means developers have control over memory allocation and deallocation. This allows for explicit control over memory operations, making it easier to simulate different scenarios for garbage collector benchmarking. Figure 8 shows the results of naive c++ implementation. We can observe the memory consumed increase as time elapse.

This is because the naive implementation does not include manual memory deallocation codes. X-axis is Elapsed time and Y-axis is Memory consumed. To check results for other conditions or details, please refer to `README.md`.

4.2.2 C++ with Reference Counting: In this project, we added C++ with *shared_ptr* version to benchmark the effect of reference counting instead of garbage collector. In C++, smart pointers[9] are used for reference counting to manage the memory of dynamically allocated objects. *std* :: *shared_ptr* provides shared ownership of dynamically allocated objects. It keeps track of the number of *std* :: *shared_ptr* instances pointing to the same object.

When the last *std* :: *shared_ptr* pointing to an object is destroyed or reset, the object is deallocated. The reference counting mechanism provided by *std* :: *shared_ptr* helps manage memory automatically, making it easier to avoid memory leaks and ensuring proper deallocation when the last owning pointer is no longer in use. However, it's crucial to be aware of potential issues like circular references, which can be addressed using *std* :: *weak_ptr*.

Figure 9 shows the results of c++ with shared pointer implementation. When an object is created, memory usage increases, but after a short period of time, if the reference counting check determines that the object is no longer used, the object is deallocated. Therefore, memory usage is reduced. Through this process, as can be seen from the graph (b), it can be seen that memory usage fluctuates significantly in the vertical direction over time.

4.2.3 C++ with Conservative Garbage Collector: This test suite is for benchmarking C++ with Conservative Garabage Collector. In this project, we added C++ with Boehm-Demers-Weiser Garbage Collector version to benchmark the effect of garbage collector in C++.

In all files, we replaced memory allocator/deallocator with the conservative GC. Boehm-Demers-Weiser Garbage Collector (BDWGC)[8] is a garbage collector for C and C++. It replaces dynamic memory allocation and deallocation with the garbage collector's functions.

This test suite is the modified code using GC_MALLOC for memory allocation and relying on the garbage collector for cleanup. The garbage collector will automatically track and collect memory that is no longer in use. We don't need to manually free memory. Figure 10 shows the results of c++

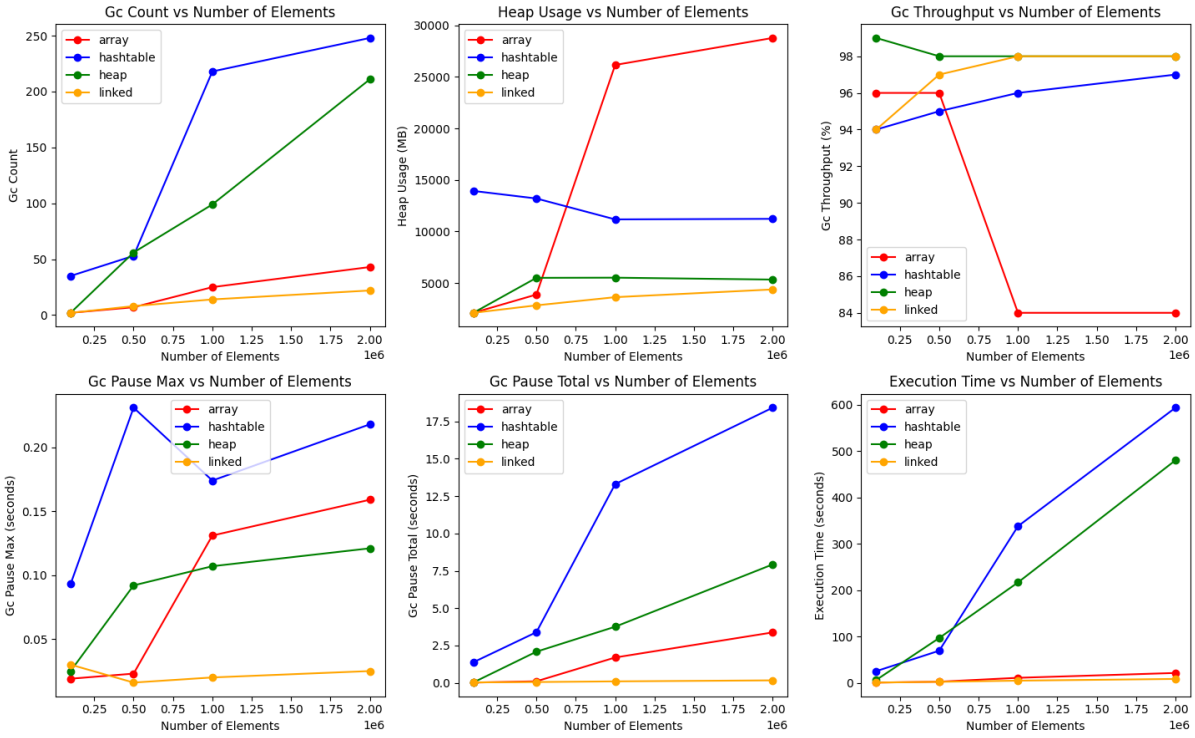

Figure 2. Java Results for Various Count of Elements

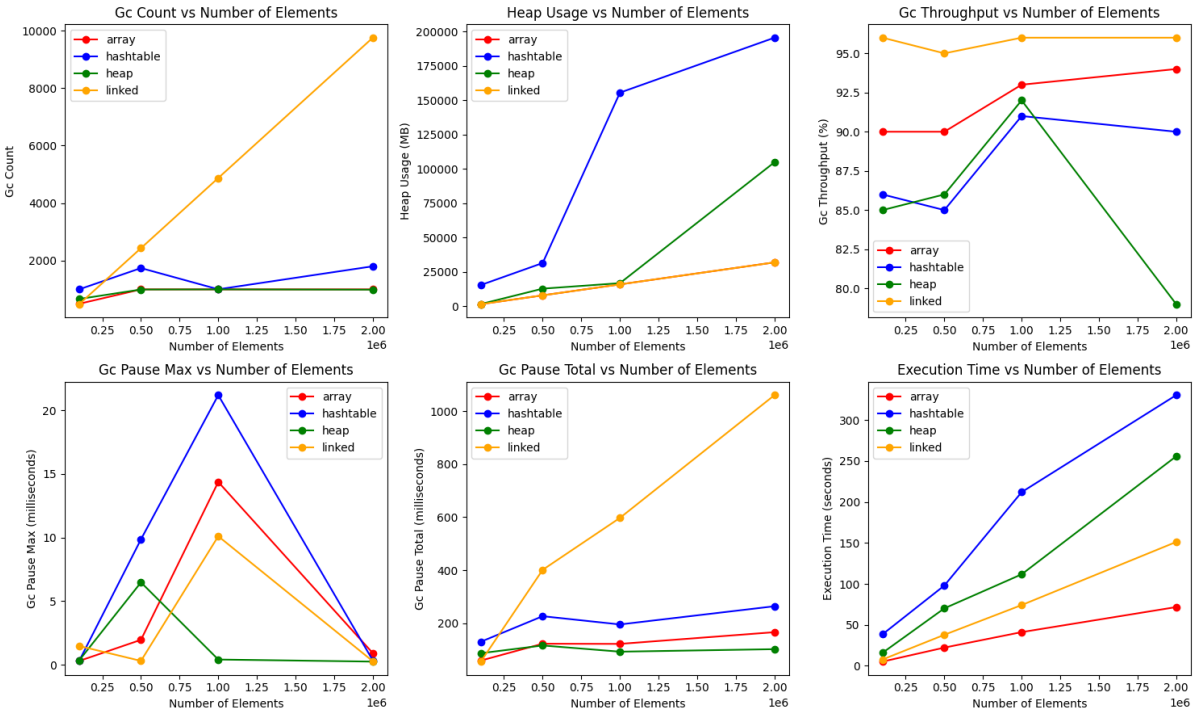

Figure 3. Golang Results for Various Count of Elements

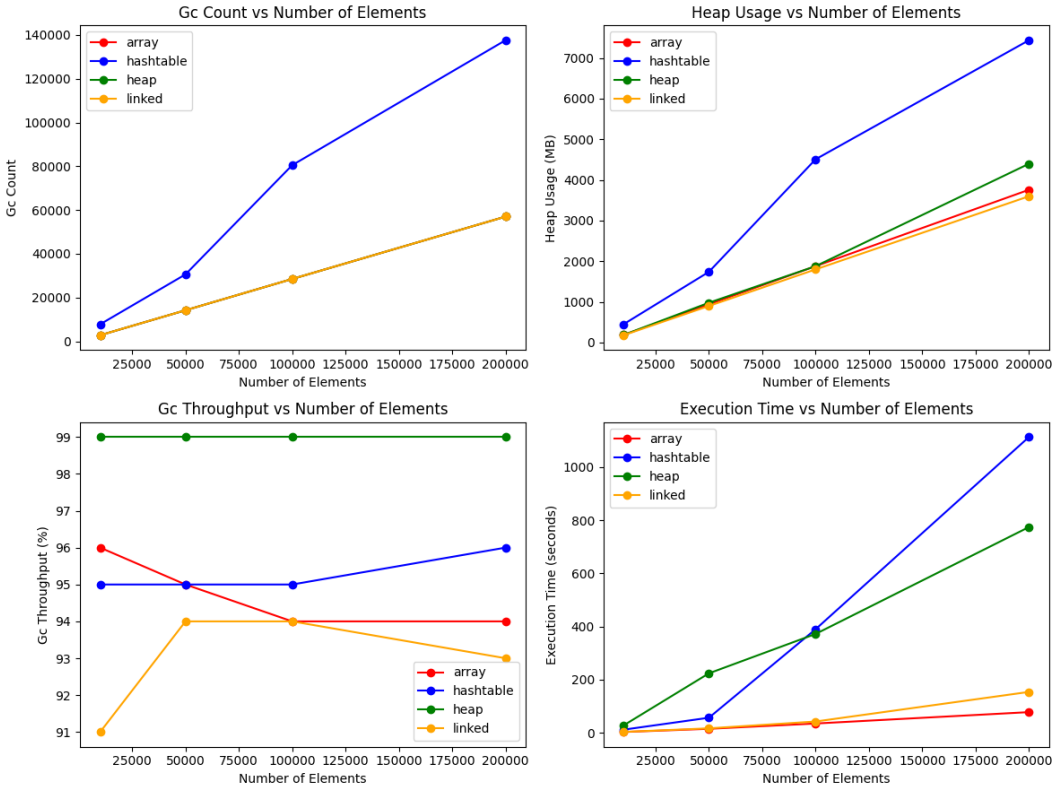

Figure 4. Python Results for Various Count of Elements

with Boehm-Demers-Weiser Garbage Collector implementation. As garbage collection is used, it can be observed that the moment the heap usage reaches a certain limit, the usage decreases sharply. The reason is that the garbage collector is triggered the moment that specific limit is reached.

4.2.4 Rust. In Rust, memory is automatically cleaned up when it goes out of scope. Rust[7] uses a system of ownership, borrowing, and lifetimes to manage memory without the need for an explicit garbage collection.

Understanding how Rust's memory management impacts the overall resource usage of a program compared to languages with garbage collectors. This can be crucial in resource-constrained environments or for applications where efficient memory usage is a priority.

Figure 11 shows the results of benchmarking done by Heaptrack tool. X-axis is Elapsed time and Y-axis is Memory consumed. To check results for other conditions or details, please refer to `README.md`.

When we tracked heap memory usage while running Rust programs, we found that at a certain limit (75 KB), the heap consumed decreases dramatically. we analyze the cause of this observation that Rust uses automatic memory management with ownership, borrowing, and lifetimes. So, if a large portion of the heap-allocated memory becomes unreachable or is deallocated, it can result in a decrease in consumed heap memory.

5 How to run the project

A `script.py` file can be found in every programming language directory (`java/`, `golang/`, `python/`). Executing this will run the benchmark for the corresponding language. As described in section 3, there are two parts of the execution:

To execute the benchmark for different number of elements, do:

```
python3 script.py elements
```

To execute the benchmark for different GC percentages, do:

```
python3 script.py gcs
```

This script will run the input parameters described in section 3 and produce a `results.csv` file. The graphs can then be generated by running the `graph.py` file in the root directory:

```
python3 graph.py <csv-file-name>
Example: python3 graph.py java/src/results.csv
```

How to execute non-gc benchmark: Rust, and three versions of C++ directories include `Makelist.txt` file. The detailed command is described in the make file. More details

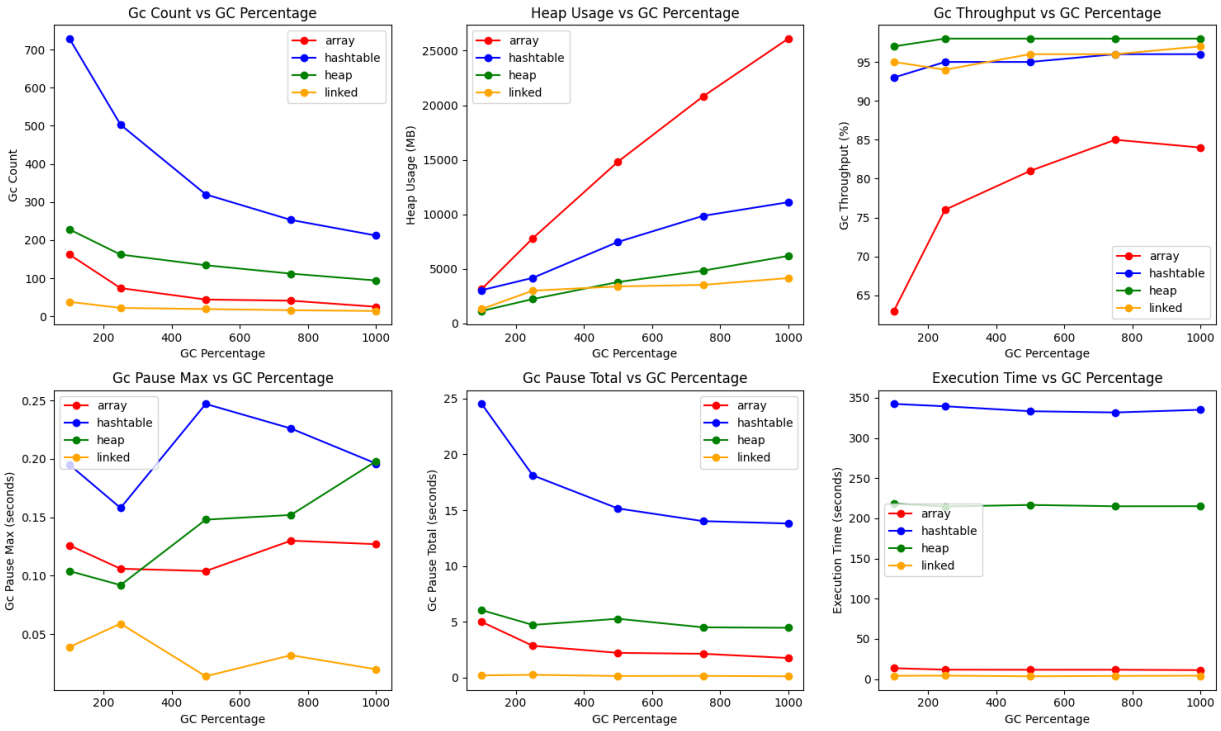

Figure 5. Java Results for Various GC Percentages

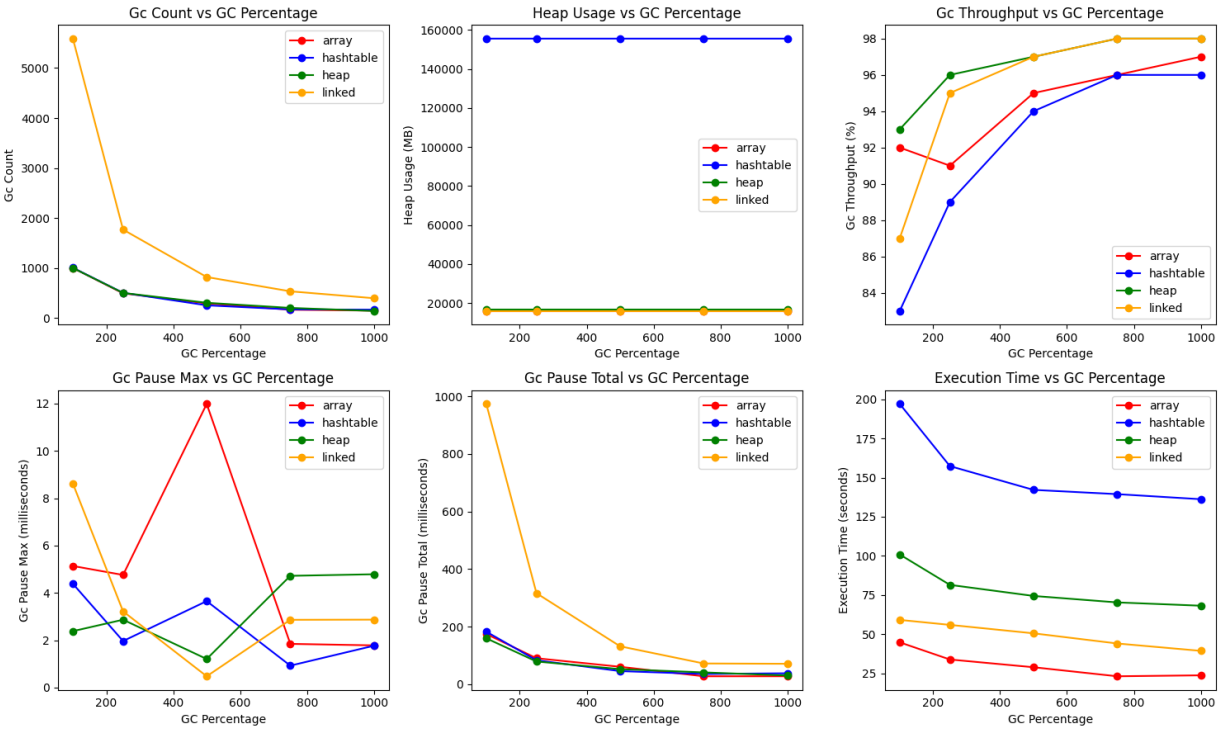

Figure 6. Golang Results for Various GC Percentages

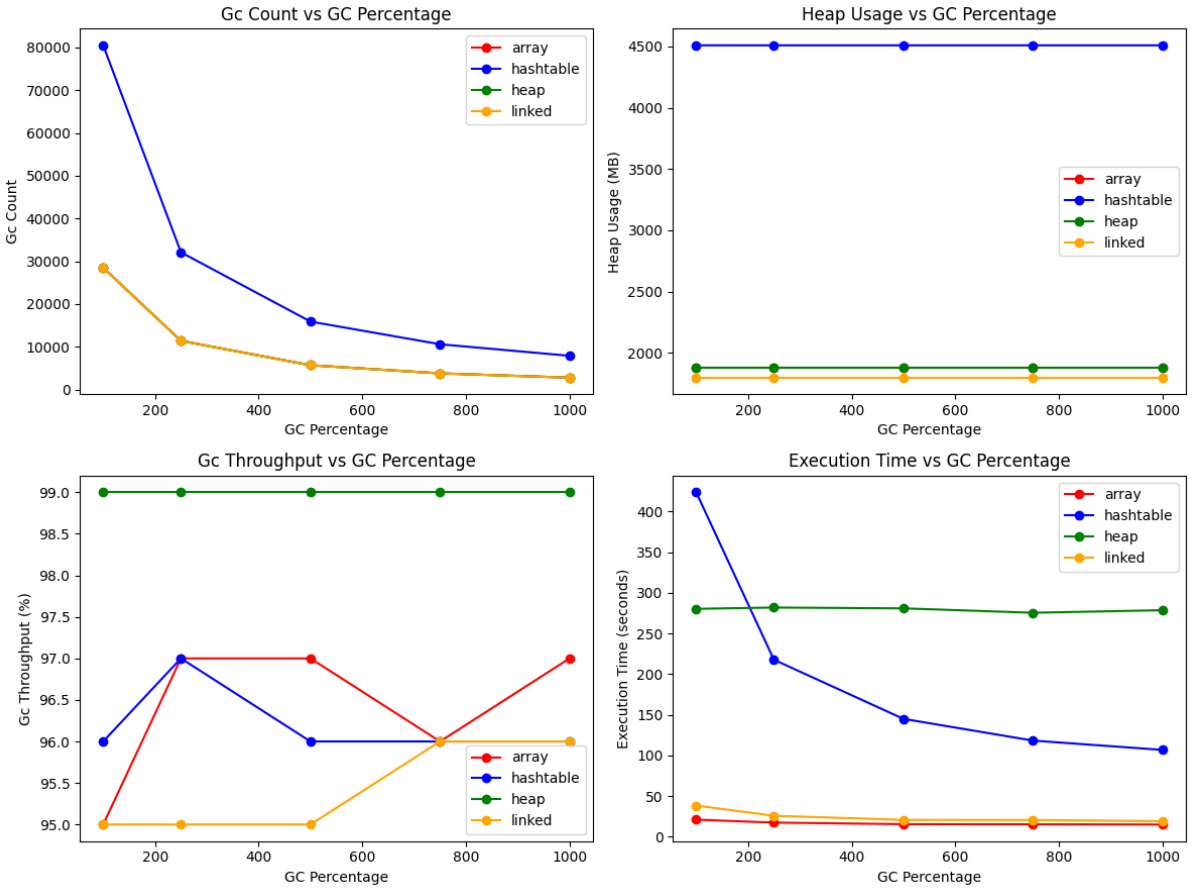

Figure 7. Python Results for Various GC Percentages

(a) Array (b) Linked List (c) Hash Table (d) Heap

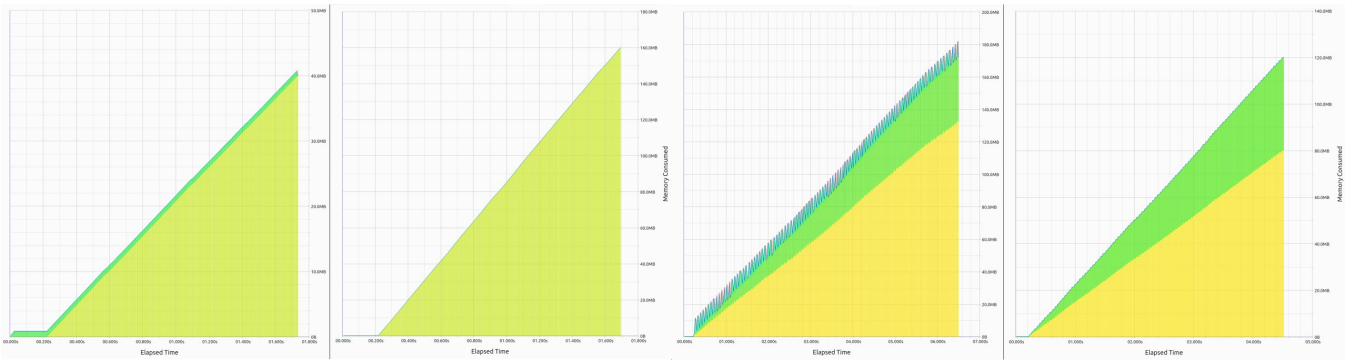

Figure 8. C++ Results for 4 Types of Data Structure

about how to execute by referring to `README.md` in each directory.

To reproduce the result for **rust** profiling, do:

```
mem.profile rust count=[count] op=[data structure name]
```

For the three types of **C++** codes, do:

```
heaptrack ./cpp_benchmark [count] [data_structure]
```

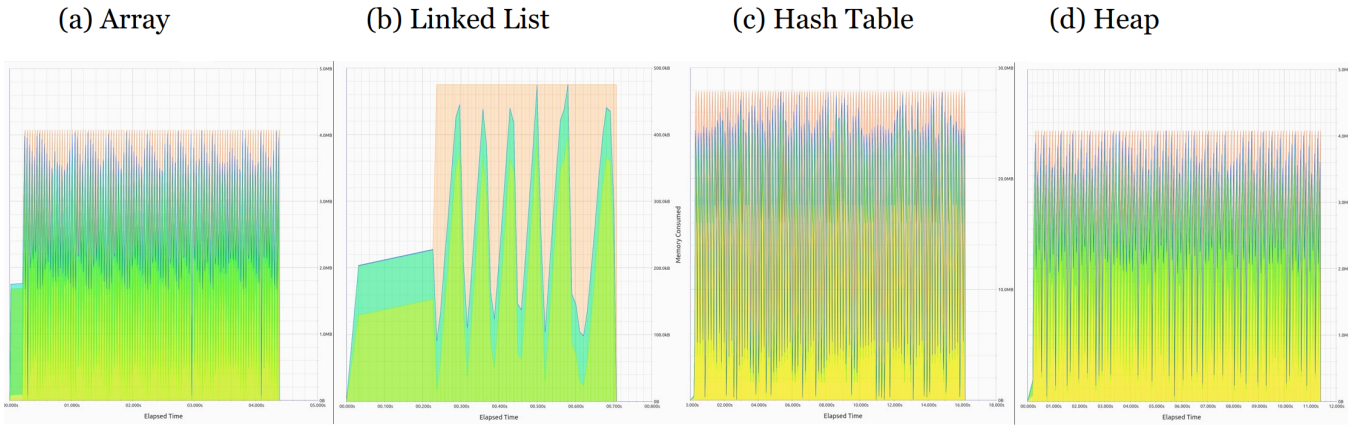

Figure 9. C++ with Reference Counting Results for 4 Types of Data Structure

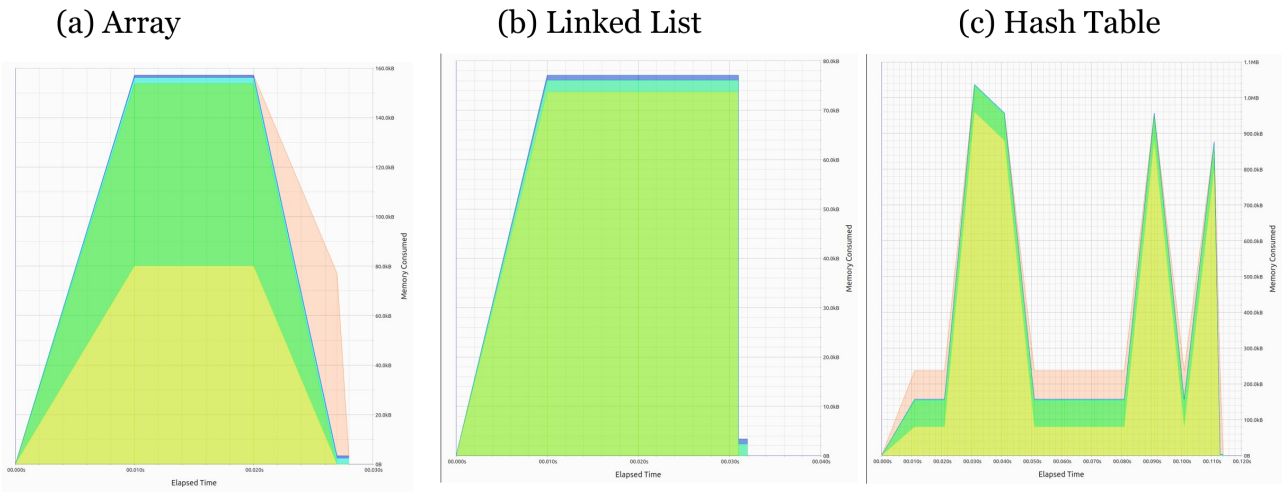

Figure 10. C++ with GC Results for 3 Types of Data Structure

6 What was learnt

6.1 Garbage collector features and its performance

This is the biggest learning experience of the entire project: garbage collector performance. Much of our initial understanding is that most garbage collectors have similar performance. But this research gave a different idea—it is not. Each programming language's garbage collector has different performance, and it shines in different use cases.

From our research, we discovered that Python has the worst GC performance of all. This is partly due to its design philosophy of reference counting mechanisms, whereby each referencing and derefencing of an object will require the interpreter to validate this, thus reducing the overall execution speed. This can be seen from our research, where we had to reduce all parameters by 10 times due to the scale amount of time taken to execute. Even with this reduction of 10 times, its performance is still incomparable with Java and Golang.

Java has the best garbage collector among Golang and Python. This could be partly because Java contains a few GC options (G1GC, CMS, Parallel GC, ZGC, etc.) employing generational, concurrent mark-and-sweep, and other algorithms, which in turn allow the language to fine-tune its performance for different needs. This explains the lowest GC counts and pause times, as shown in Figure 2.

6.2 How to design benchmark analysis

Consistent and fair benchmarking was the first priority and consideration in our design process. Most of the time in this project was spent researching the features of various programming languages' GC and deciding the metrics, approaches, and simulation style to take. As our project focuses on the performance comparison across various data structures, we decided to implement every possible common data

(a) Array (b) Linked List (c) Hash Table

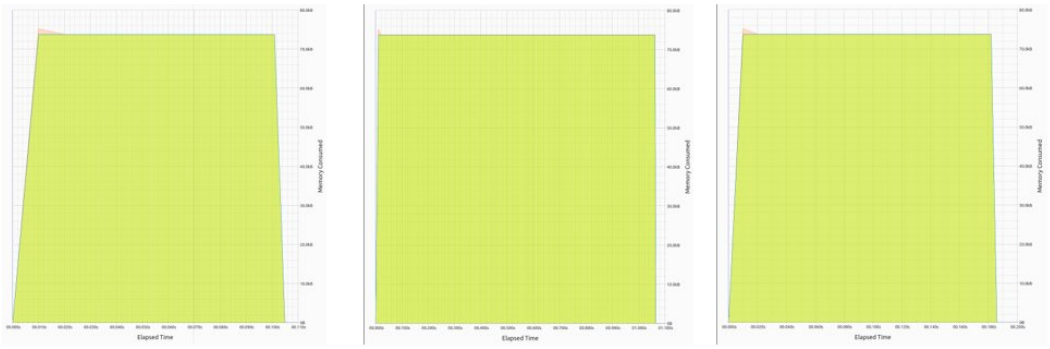

Figure 11. Rust Results for 3 Types of Data Structure

structure (such as array, list, stack, queues, linked list, binary tree, etc.). However, after a couple of experiments, we found that the difference among these data structures is not a huge one, where we could group the data structure into forms of arrays, lists, trees, and hash-based. This helped us create a generalized and broad form of benchmarking.

To create a fair benchmark, we decided to implement our own versions of data structures because the built-in data structures offered in each language could vary in implementation and design, which could in turn affect the consistency of the benchmark results. This way, it allows us to focus on manipulating variables that are only needed (such as the number of elements or GC percentage, a.k.a., heap size) in order to create a better focus analysis.

7 Conclusion

Comparing benchmark results between GC languages (Java, Go, Python) and non-GC languages (Rust, C++) was challenging due to fundamental differences in memory management strategies and runtime behaviors.

Thus, when comparing the performance, it's crucial to consider the context and the specific goals of the comparison. Each type of language has its strengths and trade-offs, and the choice often depends on factors such as application requirements, development speed, and resource constraints.

What we learned from this is that the benchmarking should be done carefully, considering the characteristics of each language and runtime environment.

References

[1] Grgic, H., Branko Mihaljević, and Aleksander Radovan. "Comparison of garbage collectors in Java programming language." 2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO). IEEE, 2018.

[2] https://github.com/mgm3746/garbagecat

[3] https://manpages.ubuntu.com/manpages/jammy/man1/heaptrack.1.html

[4] https://docs.rs/jemallocator/latest/jemallocator/

[5] https://valgrind.org/docs/manual/ms-manual.html

[6] https://uwaterloo.ca/computer-science-computing-facility/teaching-hosts

[7] https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html

[8] https://github.com/ivmai/bdwgc

[9] https://en.cppreference.com/w/cpp/memory/shared_ptr