# Part 2: Benchmarking Lua's GC

## Sub-part 1:

Using the callgrind outputs and call graphs (attached in image format in the zip) to find the percentage of instructions consumed by GC for the different GC configurations:

- Full GC: **21.54%**
- Incremental GC: **35.99%**
- Generational GC: **33.59%**

**Explanation**:

- ➔ Note that for Incremental GC and Generational GC, the function *luaC_step* was benchmarked whereas for the full GC, the function *luaC_fullgc* was benchmarked.
- ➔ The numbers show that full GC vs. incremental or generational GC involves trade-offs. While incremental or generational GC may use more instructions, it significantly reduces pause times, improving application responsiveness.
- ➔ In our tests, full GC runs only once after the function, which might not expose its full drawbacks in complex applications. In contrast, incremental and generational GC triggers more frequently and less disruptively, which is the reason why they have a higher percentage of instructions.
- ➔ Full GC can be costly in larger projects due to its complete halt of execution and less frequent memory freeing, leading to higher memory usage. Thus, the true cost of full GC, in terms of time and memory, might not be fully evident in this benchmark.

---

## Sub-part 2:

The following table indicates the percentage of instructions consumed by GC for different values of **m**, and different GC configurations. Note that the value of **n** is constant at *100*.

| | m = 100 | m = 500 | m = 1000 | m = 5000 |
|---|---|---|---|---|
| Full GC | 21.40% | 21.53% | 21.54% | 29.41% |
| Incremental GC | 31.55% | 33.54% | 35.99% | 33.55% |
| Generational GC | 28.95% | 32.51% | 33.59% | 32.87% |

**Explanation**:

- **Full GC** shows a clear increase in instruction percentage with larger **m** values due to the increasing cost of complete garbage collections.

- **Incremental GC** has variable performance, initially increasing due to overhead but slightly decreasing as **m** grows, reflecting a balance between frequent small collections and overall performance impact.
- **Generational GC** shows a similar increasing trend with some stabilization at higher **m**, indicating that the generational approach balances minor and major collections effectively but still faces increased overhead with larger data sizes.

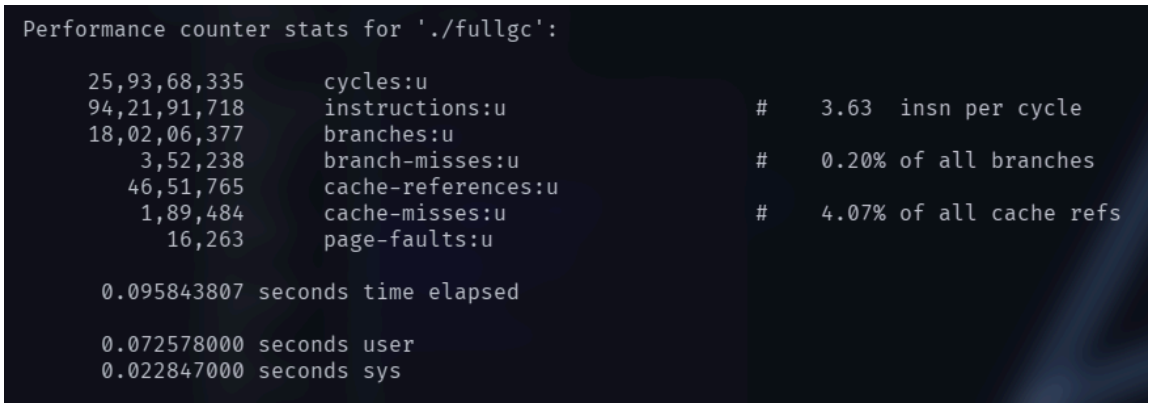
Overall, the trends suggest that while full GC becomes more costly with larger datasets, incremental and generational GCs attempt to manage memory more efficiently but also exhibit varying degrees of performance impact based on the size of input data.

---

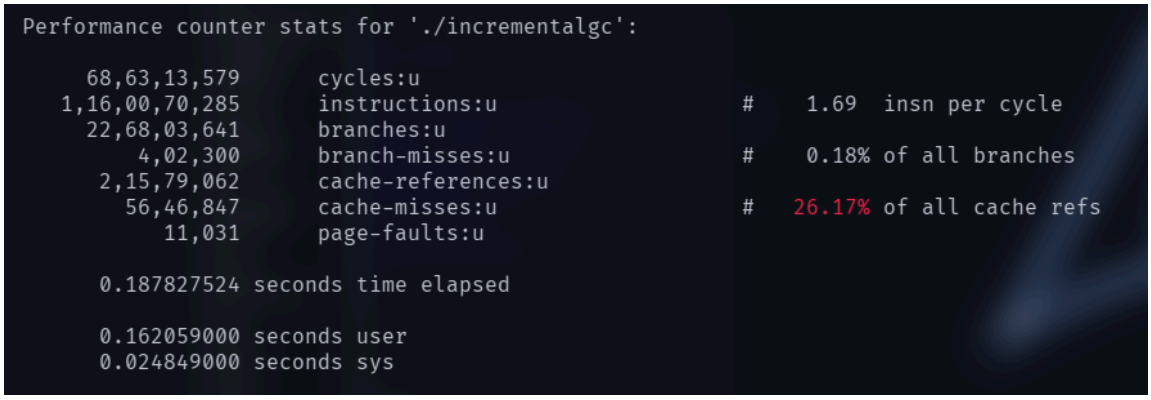
## Sub-part 3:

The following images (also included in the zip) show the results of *perf* analysis for different GC configurations including a configuration without GC.

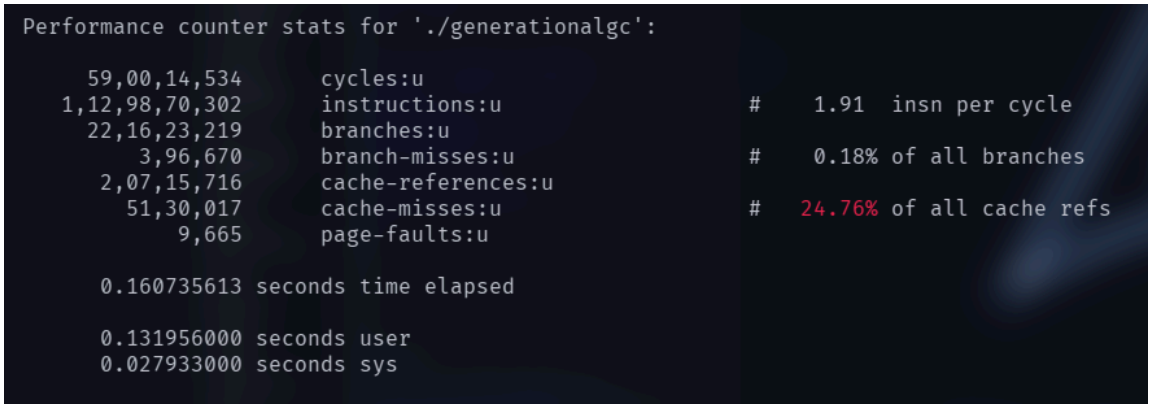
- Full GC:



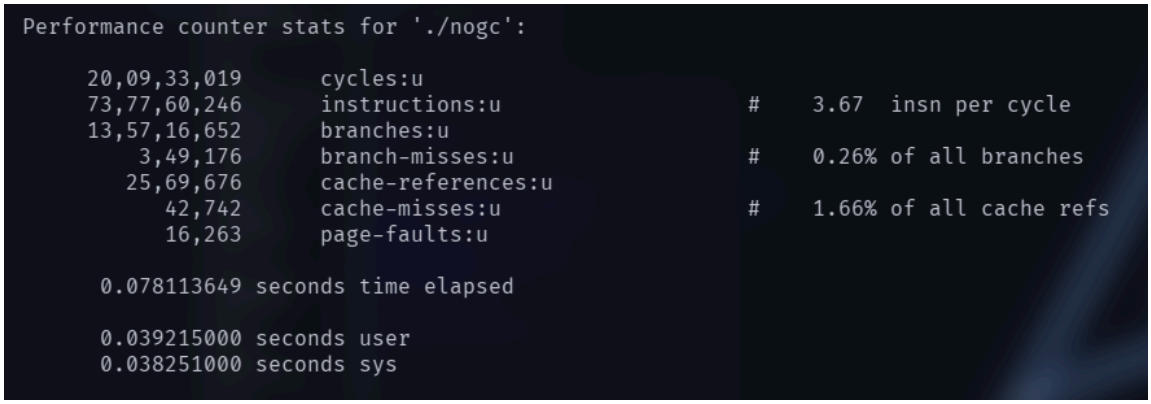

- Incremental GC:



- Generational GC:




- Without GC:




Summary of data displayed above:

| | Instructions per cycle | Branch misses | Cache misses | Page faults |
|---|---|---|---|---|
| Full GC | 3.63 | 0.20% | 4.07% | 16263 |
| Incremental GC | 1.69 | 0.18% | 26.17% | 11031 |
| Generational GC | 1.91 | 0.18% | 24.76% | 9665 |
| Without GC | 3.67 | 0.26% | 1.66% | 16263 |

**Explanation of variation in the values and discussing their relevance:**

**1. Branch Misses:**

- Relevance: Branch misses indicate how often the CPU incorrectly predicts the outcome of branches, leading to potential performance penalties. Lower branch misses suggest a more predictable control flow. However, it is not particularly very relevant here.
- Variation Explanation:
    - Incremental and Generational GC: These configurations show slightly lower branch miss rates compared to Full GC and No GC. This could be due to the

frequent but smaller steps in the incremental and generational GCs, which may reduce the complexity of the branch prediction compared to the large, infrequent pauses in Full GC.
    - No GC: Exhibits a slightly higher branch miss rate, likely due to the absence of GC logic, which results in fewer predictable patterns for the branch predictor.

**2. Page Faults:**

- Relevance: Page faults occur when the program accesses memory that is not currently mapped to physical memory, leading to potentially costly operations. This metric is relevant for understanding memory management efficiency in each GC configuration.
- Variation Explanation:
    - Incremental and Generational GC: Show fewer page faults, indicating better memory locality and management compared to Full GC. This suggests that incremental and generational GCs are more efficient in managing memory access patterns, due to their more frequent, smaller collections.
    - No GC and Full GC: Indicates that as the memory usage grows continuously, more and more page faults occur.

**3. Cache Misses:**

- Relevance: Cache misses occur when the required data is not found in the CPU cache, leading to slower memory access times. This is crucial for understanding the efficiency of memory access patterns in each GC configuration.
- Variation Explanation:
    - Incremental and Generational GC: Show significantly higher cache miss rates, which is indicative of the frequent memory allocations and deallocations that can disrupt cache locality. This high rate of cache misses highlights the trade-off between reducing pause times and maintaining efficient cache usage.
    - Full GC and No GC: Show much lower cache miss rates, with No GC having the lowest. This suggests that when the GC is either absent or performing fewer, larger collections, the memory access pattern is more cache-friendly.

**4. Instructions Per Cycle (IPC):**

- Relevance: IPC measures the average number of instructions executed per CPU cycle and is a direct indicator of CPU efficiency. Higher IPC values indicate more efficient execution.
- Variation Explanation:
    - No GC and Full GC: Both have a high IPC, indicating that in the absence of frequent GC interruptions, the CPU can execute instructions more efficiently.
    - Incremental GC: Shows the lowest IPC at 1.69, reflecting the overhead of frequent GC steps interrupting the program's execution.
    - Generational GC: Has a slightly higher IPC than Incremental GC at 1.91, which suggests that the minor collections in generational GC are less disruptive than the continual stepping of the incremental GC, but still impose a significant overhead compared to no GC.

**Conclusion:**
The performance metrics indicate that while incremental and generational GCs reduce pause times and potentially improve memory management efficiency, they come at the cost of increased cache misses and reduced IPC, leading to overall slower execution. Full GC and No GC provide more efficient CPU utilization as evidenced by the higher IPC and lower cache misses, but Full GC may lead to higher page faults due to large, infrequent collections.