Analysis of Garbage Collection in Lua

Name: Bratin Mondal
Roll Number: 21CS10016

Course Code: CS60203
Course Name: Design Optimization of Computing Systems

1 Introduction

This report discusses the variations of garbage collection algorithms used in Lua, analyzed after running them on different test benches.

2 Percentage of Instructions Consumed by Garbage Collection

Garbage Collection Algorithm	Percentage of Instructions Consumed by Garbage Collection
`full gc`	21.89%
`incremental gc`	19.87%
`generational gc`	17.08%

Table 1: Percentage of Instructions Consumed by Different Garbage Collection Algorithms

2.1 Analysis

The analysis of the results shows that the `full gc` algorithm consumes the highest percentage of instructions, followed by `incremental gc`, and finally `generational gc`.

2.1.1 Full GC

The `full gc` algorithm has the highest instruction consumption:

1. The full garbage collection was executed at the end when all memory allocations had been completed, meaning the collector had to traverse a large set of objects to determine which were garbage.
2. The `sweeplist` function, responsible for traversing and freeing garbage objects, consumed **21.50%** of the total instructions used by the garbage collector. Out of this, **18.99%** was consumed by the `freeobj` function, indicating that the collector spent most of its time freeing a large number of objects.

2.1.2 Incremental GC

The `incremental gc` algorithm consumes the second-highest percentage of instructions:

1. The incremental garbage collector operated in multiple cycles, reducing the number of objects to be traversed in each cycle.
2. The `atomic` function was executed **20** times, completing **20** full cycles of garbage collection.
3. The `singlestep` function was called **80** times, resulting in 80 pauses during execution, each completing a different step of the garbage collection process.
4. During these pauses, the `atomic` function consumed only **0.03%** of the total instructions, as the testbench had no weak tables or finalizers, which are typically handled by the `atomic` function.
5. The `propagatemark` function consumed **8.80%** of the total instructions and was called **867,609** times, with only **2,331** calls originating from the `atomic` function. This indicates that the marking task was well-distributed across cycles.
6. The `sweepstep` function consumed **8.78%** of the total instructions, with **8.76%** used by the `sweeplist` function. This is lower than the `full gc` algorithm's consumption, as the incremental collector traversed a smaller set of objects in each cycle.

2.1.3 Generational GC

The `generational gc` algorithm is the most efficient in terms of instruction consumption:

1. The generational garbage collector, similar to the incremental one, operated in multiple cycles, further reducing the objects traversed per cycle.
2. The `luaC_step` function was called **22** times, indicating **22** pauses for garbage collection.
3. The `sweepgen` function was called **30** times, completing **6** young generational collections (5 calls are made for each young generational collection) only traversing the young generation objects. This optimized approach reduced the instruction consumption compared to the incremental garbage collector.
4. The `atomic` function was called **21** times, consuming **8.30%** of the total instructions, higher than the incremental garbage collector. This difference arises because the generational collector checks the whole object set during major collections, while the incremental collector only traverses the gray objects.
5. The `propagatemark` function consumed **7.97%** of the total instructions and was called **788,835** times, with **788,786** calls from the `atomic` function, indicating a significant amount of marking done during each pause.
6. The `sweepstep` function consumed **8.75%** of the total instructions, with **8.73%** used by the `sweeplist` function. This is much lower than the `full gc` and slightly lower than the `incremental gc`, as the generational collector also traversed fewer objects per cycle.

2.2 Conclusion

The analysis reveals that the generational and incremental garbage collectors are more efficient than the full garbage collector. The generational collector has a slight edge due to its separation of objects into young and old generations, allowing for frequent minor collections and reducing instruction consumption. The incremental garbage collector, while also efficient, consumes slightly more instructions due to its multi-cycle approach. The full garbage collector is the least efficient, as it must traverse the entire object set in a single cycle, leading to higher instruction consumption.

3 TestBench with Different Parameters

m,n	Full GC	Incremental GC	Generational GC
100,100	21.76%	16.64%	15.07%
500,100	21.87%	18.48%	15.48%
1000,100	21.89%	19.87%	17.08%
5000,100	27.81%	15.03%	18.43%

Table 2: Percentage of Instructions Consumed by Garbage Collection in Different TestBenches

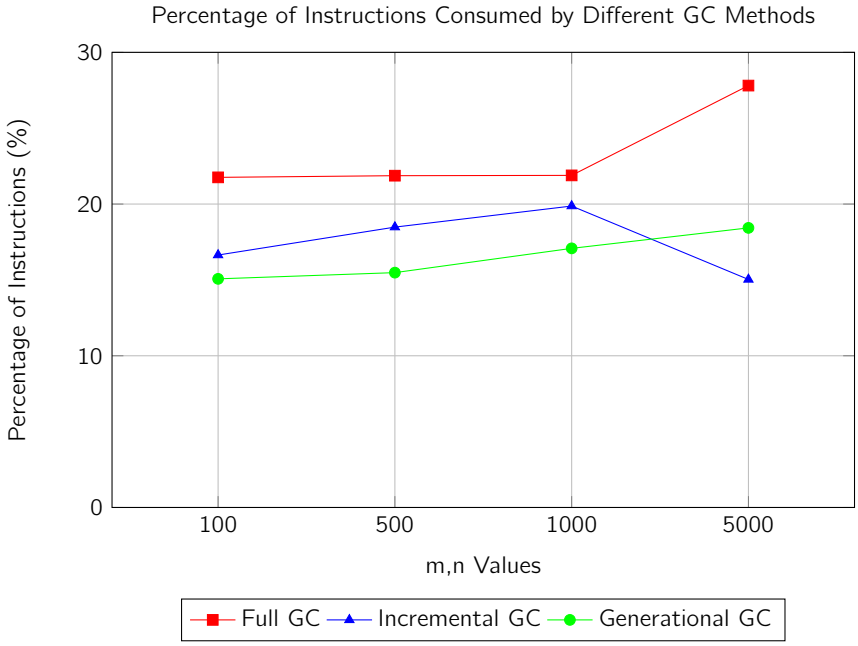

Figure 1: Percentage of Instructions Consumed by Different Garbage Collection Methods in Various TestBenches.

We analyze the percentage of instructions consumed by garbage collection in different testbenches with varying `m,n` values for all three garbage collection algorithms.

3.1 Analysis

3.1.1 Full GC

- Full GC consistently consumes the highest percentage of instructions compared to the other garbage collection methods.
- From testbench 1 to testbench 3, Full GC's instruction consumption increases slightly. This indicates that as the matrix size grows, Full GC starts to consume a slightly higher percentage of instructions. However, for lower values of `m,n`, the difference is negligible, suggesting stability over a smaller range of matrix sizes.
- In testbench 4, Full GC's instruction consumption increases significantly to **27.81%** from **21.89%** in testbench 3. This sharp increase highlights that the overhead for Full GC grows significantly with larger matrix sizes, likely due to the increased amount of memory and objects being managed.

In essence, Full GC is manageable for smaller to moderately sized datasets, but its cost escalates significantly with larger datasets. This is the main reason why Full GC is not suitable for real-time applications or large-scale systems.

3.1.2 Incremental GC

- Incremental GC consumes a moderate percentage of instructions, performs second-best on testbenches 1 to 3 and best on testbench 4.
- In testbench 1, Incremental GC consumes **16.64%** of instructions. There were **14** calls to the `luaC_step` function and **14** calls to the `atomic` function. This indicates a total of **14** cycles of garbage collection. Although Incremental GC performed better than Full GC, the distribution of marking and sweeping tasks was not optimally managed across pauses.
- In testbench 2, Incremental GC consumes **18.48%** of instructions. The increase in percentage is mainly due to managing a larger amount of memory and objects. The number of calls to `luaC_step` increased significantly to **42**, while calls to the `atomic` function increased slightly to **18**. This suggests that marking and sweeping tasks were not perfectly distributed across pauses. Despite the higher instruction consumption compared to testbench 1, Incremental GC better managed the increased matrix size by distributing tasks across pauses.
- In testbench 3, Incremental GC consumes **19.87%** of instructions. The number of calls to `luaC_step` increased to **80**, and calls to the `atomic` function increased to **20**. While the percentage of instructions consumed rose slightly compared to testbench 2, the distribution of marking and sweeping tasks was improved.
- In testbench 4, Incremental GC consumes **15.03%** of instructions. This is the lowest percentage of instructions consumed by Incremental GC across all testbenches, and it represents the best performance among all garbage collection algorithms for this testbench. The number of calls to `luaC_step` increased to **276**, while calls to the `atomic` function increased only to **23**. This indicates that Incremental GC managed the increased workload more efficiently as the matrix size grew.

Incremental GC aims to interleave garbage collection steps with program execution using tri-color marking schemes and barriers. This approach updates the status of objects efficiently, reducing the time spent on marking by only traversing gray objects managed by barriers. For larger datasets, Incremental GC demonstrates improved performance compared to other garbage collection algorithms.

3.1.3 Generational GC

- Generational GC performs best on testbenches 1 to 3, consuming the least percentage of instructions among all GC types, and second-best on testbench 4, where Incremental GC outperforms it slightly.
- The percentage of instructions consumed by Generational GC increases slightly from testbench 1 to testbench 2, rising from **15.07%** to **15.48%**. However, a more significant increase is observed from testbench 2 to testbench 3, where it jumps from **15.48%** to **17.08%**. This upward trend continues from testbench 3 to testbench 4, where the consumption rises from **17.08%** to **18.43%**. This pattern suggests that Generational GC is highly efficient for smaller matrices, but its efficiency diminishes as the matrix size grows due to the increased complexity in managing larger memory footprints.
- Although Generational GC is designed to optimize memory management by segregating objects into young and old generations, it struggles with larger matrices. As the number of objects in the old generation increases, the collector needs to perform more extensive and frequent collections, which involve traversing a large number of objects. This comprehensive traversal becomes more costly, leading to a noticeable decline in performance on testbench 4 compared to the incremental garbage collector.
- Larger matrices increase the memory footprint, leading to more frequent and intensive GC activity. As the workload grows, managing the old generation becomes more demanding, resulting in higher instruction consumption and reduced GC efficiency.

Generational GC is efficient for smaller datasets but struggles with larger matrices due to the increased complexity of managing larger memory footprints. The generational approach is designed to optimize memory management by segregating objects into young and old generations, but it becomes less effective as the workload grows, leading to higher instruction consumption and reduced efficiency.

3.2 Conclusion

The analysis shows that Incremental GC is the most efficient for larger datasets, consuming the least instructions in testbench 4. Generational GC performs well with smaller datasets but becomes less efficient as matrix size increases, leading to higher instruction consumption. Full GC is the least efficient, with significant instruction costs for larger datasets. Incremental GC's interleaved approach and tri-color marking schemes make it better suited for managing larger workloads compared to other GC algorithms. While Generational GC excels with smaller datasets, its efficiency declines with growing workloads.

4 Analysis with Perf

The performance analysis of garbage collection algorithms using the `perf` tool is shown in Table below. The readings are average of 100 runs.

Garbage Collection Algorithm	Branch Misses	Page Faults	Cache Misses	Instructions per Cycle
`no gc`	421,607	16,258	1,581,617	3.08
`full gc`	413,775	16,258	1,582,313	3.04
`incremental gc`	503,439	11,131	6,413,636	2.04
`generational gc`	486,692	10,088	7,671,638	1.89

Table 3: Performance Analysis of Garbage Collection Algorithms

4.1 Branch Misses

4.1.1 Importance

1. **Impact on Execution Flow:** GC algorithms often involve complex memory operations that can lead to unpredictable execution paths, challenging the branch predictor and potentially degrading performance.
2. **Effect on Control Flow Predictability:** Frequent changes in the approach of the garbage collector during GC operations can reduce the accuracy of the branch predictor, increasing branch miss rates and leading to inefficiencies in CPU execution.
3. **Correlation with Algorithm Complexity:** More complex GC algorithms with intricate memory management and object traversal logic tend to disrupt the branch predictor more frequently, resulting in higher branch miss rates and reduced performance.

4.1.2 Analysis

1. **No GC:**
   - **Branch Misses:** 421,607
   - **Explanation:** The baseline measurement for branch misses shows minimal branch mispredictions due to stable execution paths. However, the branch miss rate is higher than in the Full GC scenario, possibly due to the `lua_close` function's approach to freeing resources, which may introduce some unpredictability in the control flow.
2. **Full GC:**
   - **Branch Misses:** 413,775
   - **Explanation:** Full GC shows a slight reduction compared to No GC. This is due to Full GC pausing the entire program for garbage collection only at the end. The collector's execution pattern during this pause is relatively stable and predictable, leading to fewer branch mispredictions, even lower than the baseline.
3. **Incremental GC:**
   - **Branch Misses:** 503,439
   - **Explanation:** Incremental GC experiences the highest branch miss rate. This algorithm performs garbage collection in small, interleaved steps, leading to frequent switching between program execution and GC operations. This results in high unpredictability and more frequent branch mispredictions.
4. **Generational GC:**
   - **Branch Misses:** 486,692

- **Explanation:** Generational GC shows fewer branch misses than Incremental GC but more than Full GC. The complexity of managing object generations and making decisions about which objects to collect or promote adds some unpredictability to execution paths.

4.2 Page Faults

4.2.1 Importance

1. **Memory Management:** Efficient garbage collection that reuses memory can reduce page faults and improve performance.
2. **GC Impact on Memory:** Frequent memory access during GC operations can lead to increased page faults, especially if the GC algorithm does not consider memory locality.
3. **Performance Implications:** Excessive page faults can degrade performance significantly due to slower disk I/O operations. Optimizing GC to minimize page faults is crucial for maintaining performance.

4.2.2 Analysis

1. **No GC:**
   - **Page Faults:** 16,258
   - **Explanation:** The baseline scenario with no garbage collection shows a moderate level of page faults due to the application's own memory demands and system paging behavior.
2. **Full GC:**
   - **Page Faults:** 16,258
   - **Explanation:** Full GC maintains the same page fault rate as No GC, as it involves stopping the entire application only at the end of the process, without significantly altering memory access patterns.
3. **Incremental GC:**
   - **Page Faults:** 11,131
   - **Explanation:** Incremental GC results in fewer page faults compared to No GC and Full GC, indicating effective memory management by performing garbage collection in small steps that maintain memory locality.
4. **Generational GC:**
   - **Page Faults:** 10,088
   - **Explanation:** Generational GC has the lowest page fault rate, showing effective memory management by segregating objects into generations and frequently collecting young objects to minimize paging.

4.3 Cache Misses

4.3.1 Importance

1. **Memory Access Patterns:** GC algorithms that traverse large memory regions or have complex heuristics can lead to increased cache misses, unable to use the caching system effectively.
2. **Performance Degradation:** Poor memory access patterns during GC can lead to increased cache misses, affecting overall performance.

4.3.2 Analysis

1. **No GC:**
   - **Cache Misses:** 1,581,617
   - **Explanation:** The baseline performance with no garbage collection shows relatively low cache misses, indicating efficient memory access without GC overhead.
2. **Full GC:**
   - **Cache Misses:** 1,582,313
   - **Explanation:** Full GC introduces a slight increase in cache misses compared to No GC. This increase is minimal, reflecting that the pause for GC may not significantly impact cache performance.
3. **Incremental GC:**
   - **Cache Misses:** 6,413,636
   - **Explanation:** Incremental GC results in a significant increase in cache misses due to frequent switching between GC and program execution and accessing different memory regions during small GC steps which invalidates cache lines.
4. **Generational GC:**
   - **Cache Misses:** 7,671,638
   - **Explanation:** Generational GC shows the highest number of cache misses due to frequent object promotion and the need to manage multiple generations, leading to extensive memory traffic outside the cache and incurs most cache misses.

4.4 Instructions per Cycle (IPC)

4.4.1 Importance

1. **Impact of GC Overhead:** GC algorithms can introduce overhead that affects the CPU's ability to execute instructions efficiently avoiding stalls.
2. **Performance Bottlenecks:** Lower IPC can signal performance bottlenecks caused by GC operations, such as increased branch mispredictions or cache misses.

4.4.2 Analysis

1. **No GC:**
   - **IPC:** 3.08
   - **Explanation:** Without GC, the system achieves the highest IPC of 3.08, reflecting optimal CPU efficiency with no GC-induced overhead. The baseline performance is limited only by code execution and hardware capabilities.
2. **Full GC:**
   - **IPC:** 3.04

- **Explanation:** The IPC for Full GC is slightly lower than No GC, indicating a minor reduction in CPU efficiency due to the garbage collection overhead. The pause for Full GC at the end of the process introduces some overhead, leading to a small decrease in IPC compared to the baseline.

3. **Incremental GC:**
   - **IPC:** 2.04
   - **Explanation:** Incremental GC significantly lowers IPC to 2.04 due to the overhead of managing garbage collection in small, incremental steps. This results in increased branch mispredictions and cache misses, reducing overall CPU efficiency.
4. **Generational GC:**
   - **IPC:** 1.89
   - **Explanation:** Generational GC exhibits the lowest IPC of 1.89. The significant reduction in IPC is due to the high overhead associated with managing multiple generations of objects, leading to extensive memory traffic and inefficient CPU utilization.