Part 1: Understanding Lua's Garbage Collector

Introduction:

Lua's garbage collection (GC) mechanism is crucial for memory management, ensuring that unused memory is efficiently reclaimed. The garbage collector (GC) in Lua can work in two modes: incremental and generational. However, to run a full GC, we can manually stop the default GC before execution of test bench code, then execute a full collect cycle. This report presents a comparative analysis of these modes by leveraging a testbench script that creates a large matrix in Lua, executing it under different GC configurations. The performance and behavior of each mode are benchmarked using Callgrind and memusage.

Configurations and Execution:

To understand Lua's GC, three C programs were written— *fullgc.c*, *incrementalgc.c*, and *generationalgc.c*. Each program was designed to load and execute the provided Lua testbench script under a specific GC configuration. Here's a brief overview of each mode:

- **Full GC**: This traditional stop-the-world method halts the entire program during garbage collection. While simple, it can cause noticeable pauses in program execution, especially with large heaps.
- **Incremental GC**: This is Lua's default GC mode. It interleaves garbage collection with program execution, performing the collection in small steps. This approach minimizes pauses but may result in more frequent, smaller collections.
- **Generational GC**: This mode is based on the observation that most objects die young. It segregates objects into younger and older generations, collecting younger generations more frequently while collecting older ones less often. This can improve performance by reducing the amount of memory traversed in most GC cycles.

Callgraph Analysis:

Using Callgrind (a tool available with Valgrind), the execution of each GC mode was profiled, and call graphs were generated to visualize the behavior of Lua's GC (attaching part 2 of the submission). The following observations were made:

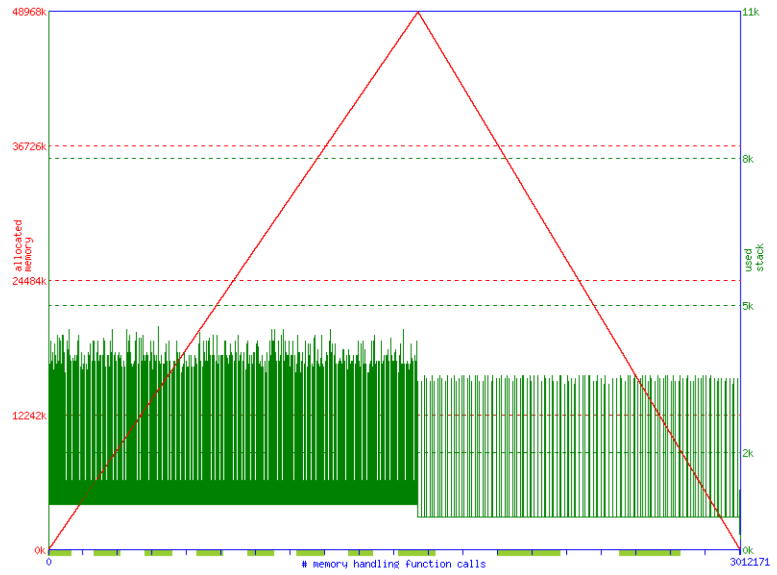
- **Overall call graph Structure**: The call graphs for full, incremental and generational GC modes were found to be very similar, with nearly identical numbers of instruction fetches (IR calls). This similarity suggests that all the modes execute the core Lua code in a comparable manner, with the primary difference lying in how and when memory is reclaimed.
- **Call Frequency of the function responsible for allocation and deallocation**: A significant difference was observed in the number of calls to the *l_alloc.lto_priv.0* function between full, incremental and generational GC modes. This function majorly calls the *free* and *realloc* functions, which are both used to free or use unused memory. The full GC mode exhibited a much higher IR calls than the generational GC mode mode, which inturn had higher IR calls than the incremental GC mode. This is consistent with the behavior of GC modes. The full GC frees the entire memory in one big chunk, whereas in generational GC, memory is freed in large chunks when older generations are

collected. In contrast, the incremental GC's stepwise approach results in fewer immediate *free* operations, spreading them out over the program's execution.

Memusage Analysis:

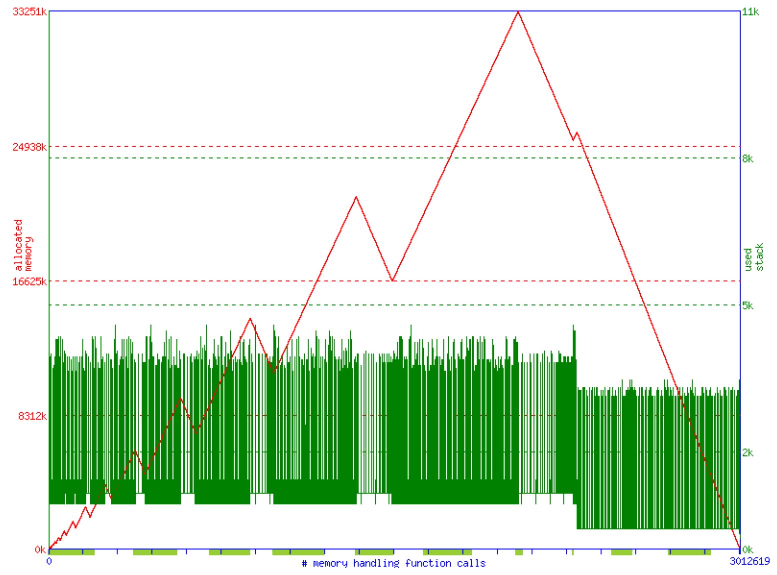
Using Memusage, the memory usage of each GC mode was profiled, for both stack and heap memory. Hoverver, we are concerned with only heap memory, which is dynamically allocated. The red lines indicate the heap usage. The following observations were made:

i) Full GC:



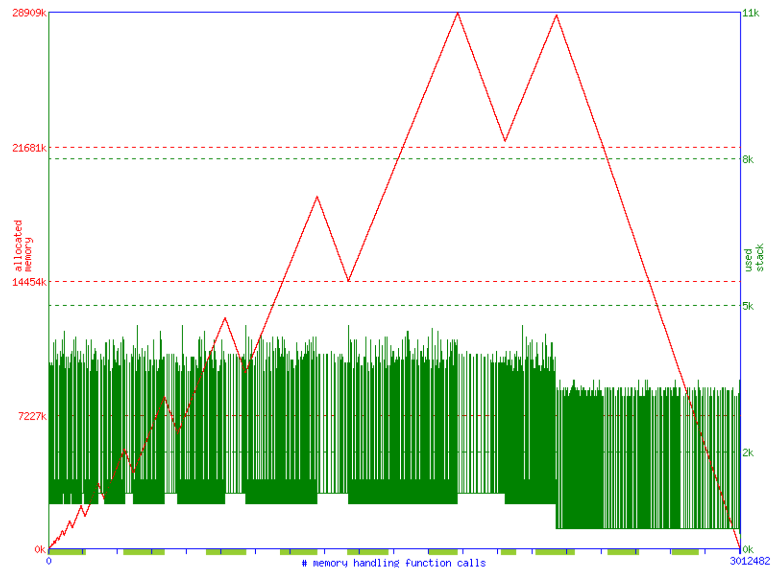
Here, the memory is constantly allocated until the full GC is triggered once, after which almost all of it is freed. Hence, a simple triangular graph is observed. The peak memory usage is about 48MBytes, which is inline with the estimation of 50MBytes.

ii) Incremental GC



Here, the peak memory usage is reduced drastically as the incremental GC continuously frees up memory. The same is observed in the graph in the form of drops in memory usage. Note the collector waits for the total memory in use to double before starting a new garbage collection cycle, which is very much expected according to the documentation.

iii) Generational GC:



Here, the peak memory usage is further reduced as the generation GC frees up memory for the short lived objects more frequently. It is more aggressive in nature, and needs more resources. Note that theoretically, the peak heap memory usage should be about 26MBytes, as calculated from the testbench code.

Conclusions:

- **Incremental GC**: Balances performance and responsiveness by spreading GC work across multiple steps. While it reduces the impact of any single GC cycle, it does not always free memory as aggressively as generational GC.
- **Generational GC**: Optimized for programs with many short-lived objects, it showed more frequent memory deallocations via the *free* function. This can lead to improved performance in scenarios with high object turnover, though it may introduce more frequent GC pauses when older generations are collected.
- **Full GC**: While simple and effective for certain workloads, it is less suited for real-time or interactive applications due to its stop-the-world nature. However, it remains a viable option when predictability and simplicity are prioritized over performance.

The insights gained from this analysis highlight the trade-offs between different garbage collection strategies in Lua. Incremental GC offers a middle ground, while generational GC is more aggressive in memory reclamation at the cost of potential performance overhead during older generation collections. Full GC, while less complex, can introduce significant execution delays, making it less suitable for performance-critical applications.