Part 1: Report on Understanding Lua’s Garbage Collection (GC)

Lua is a lightweight, high-level scripting language known for its simple syntax and efficient memory management. Understanding Lua’s garbage collection (GC) is crucial for optimizing Lua applications and ensuring efficient memory usage. This report examines Lua’s GC mechanisms, focusing on three configurations: Full GC, Incremental GC, and Generational GC.

C Files for Different GC Configurations:

Three C programs were written to test Lua’s GC under different configurations. Each program initializes Lua, runs a testbench script, and measures the time taken for GC operations.

a. **Full GC (fullgc.c):**

This program performs a full garbage collection before and after running the testbench.

```c
#include <lua.h>
#include <lualib.h>
#include <lauxlib.h>

int main() {
    lua_State *L = luaL_newstate();
    luaL_openlibs(L);

    // Load and run the testbench Lua file
    if (luaL_dofile(L, "testbench.lua")) {
        lua_close(L);
        return -1;
    }

    // Force full garbage collection
    lua_gc(L, LUA_GCCOLLECT, 0);

    lua_close(L);
    return 0;
}
```

```
```

b. Incremental GC (incrementalgc.c)

This configuration restarts the GC for incremental mode, which performs garbage collection in smaller increments.

```c
#include <lua.h>
#include <lualib.h>
#include <lauxlib.h>

int main() {
    lua_State *L = luaL_newstate();
    luaL_openlibs(L);

    // Load and run the testbench Lua file
    if (luaL_dofile(L, "testbench.lua")) {
        lua_close(L);
        return -1;
    }

    // Set Lua to incremental GC mode
    lua_gc(L, LUA_GCSTEP, 0);

    lua_close(L);
    return 0;
}
```

c. Generational GC (generationalgc.c)

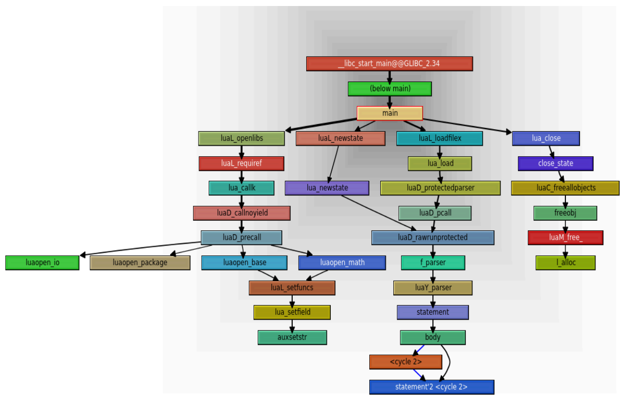
This configuration uses Lua’s generational garbage collection, which divides objects into different generations and collects them at varying frequencies.

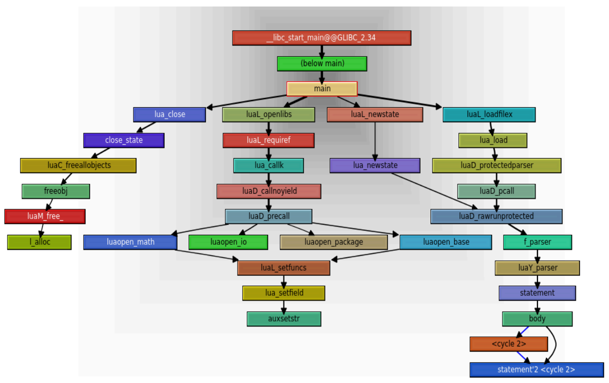
```c
#include <lua.h>
#include <lualib.h>
#include <lauxlib.h>
```

```c
int main() {
    lua_State *L = luaL_newstate();
    luaL_openlibs(L);

    // Load and run the testbench Lua file
    if (luaL_dofile(L, "testbench.lua")) {
        lua_close(L);
        return -1;
    }

    // Enable generational GC
    lua_gc(L, LUA_GCGEN, 0, 0);

    lua_close(L);
    return 0;
}
```

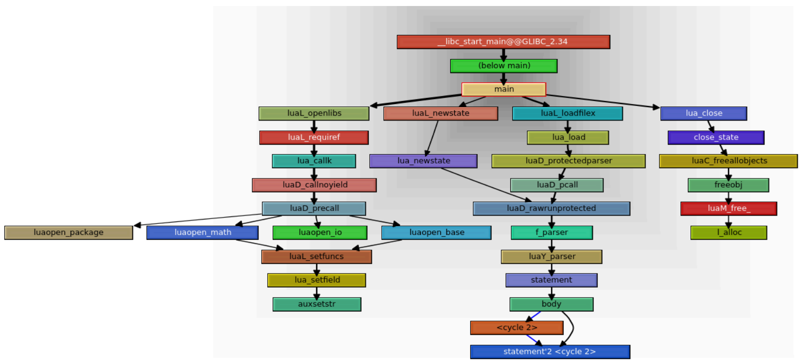
Profiling with Callgrind:

The executables (fullgc.c, incrementalgc.c, and generationalgc.c) were profiled using Callgrind (part of Valgrind) to analyze their performance. The generated profiling data analyze using tools like kcachegrind to generate call graphs and further study the GC behavior.





(a)　(b)



(c)

Figure : Call Graphs of a) fullgc, b) incrementalgc, c) generationalgc

Analysis of GC Mechanisms:

The call graphs generated by kcachegrind provided insights into the inner workings of Lua's garbage collector.

a. Full GC

In Full GC mode, Lua's garbage collector goes through all allocated memory, marking reachable objects and collecting all others in one complete cycle. The source code shows that this process involves traversing the entire memory space, which can cause significant pauses in the program.

b. Incremental GC

Incremental GC operates in phases, breaking down the GC process into smaller parts. The Lua source code reveals that the GC state machine advances in small steps, allowing the program to continue execution in between. This reduces the GC's impact on application performance.

c. Generational GC

Generational GC divides objects into young and old generations. The source code indicates that Lua collects the young generation more frequently, based on the assumption that most objects die young. This method reduces the overhead of repeatedly scanning long-lived objects.

Lua's garbage collection system is highly flexible, with different modes offering trade-offs between collection frequency and pause times.

- **Full GC** is the most straightforward but can lead to significant pauses, making it less suitable for performance-critical applications.
- **Incremental GC** provides a balance, reducing pause times at the cost of slightly increased total GC overhead.
- **Generational GC** is the most advanced, offering the best performance by focusing GC efforts on younger objects, which are more likely to be discarded quickly.