Study of Garbage Collection in Lua

Name: Bratin Mondal
Roll Number: 21CS10016

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

1 Introduction

Lua, a dynamically typed language, originally used a basic mark-and-sweep garbage collector. Lua 5.1 introduced an incremental collector to improve performance by interleaving collection with program execution. Although Lua 5.2's generational collector aimed to optimize memory management, it was reverted in 5.3 due to challenges. The generational collector was reintroduced with enhancements in Lua 5.4. This report examines the evolution and implementation of Lua's garbage collection mechanisms.

2 Fundamentals of Lua's Garbage Collection

To understand Lua's garbage collectors, we must first grasp the basics of its memory management. Lua, written in C, manages memory through objects of eight basic types: `nil`, `boolean`, `number`, `string`, `function`, `userdata`, `thread`, and `table`. These objects are typically organized into type-specific lists for memory management.

2.1 Root Set

In Lua's garbage collection mechanism, only objects that are accessible from the root set are preserved during a garbage collection cycle. The root set comprises objects that are directly reachable and thus considered live. In Lua, the root set includes:

- The registry, which holds the global table (_G), the main thread, and the `package.loaded` table that tracks loaded modules.
- Shared metatables, which are used across various parts of the Lua environment.

2.2 Tri-Color Marking

Lua's garbage collector marks objects with three colors: white, gray, and black.

- **White**: Objects not reachable from the root set.
- **Gray**: Objects reachable from the root set but with unmarked references.
- **Black**: Objects reachable from the root set with all references marked.

Lua uses two types of white objects, `WHITE0` and `WHITE1`, to differentiate between objects marked in the current and previous cycles. The main invariant is that no black object can reference a white object, with gray objects forming the boundary between them. Gray objects must be revisited before the cycle ends and kept in the `gray` list. Objects demoted from black to gray are added to the `grayagain` list for revisiting during the atomic [3.4] phase.

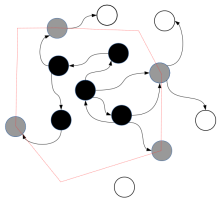

Figure 1: Tri-Color Marking

2.3 Age-Based Marking

In Lua's generational garbage collector, objects are assigned an age along with their color, which is updated after each cycle. The age categories are:

G_NEW denotes objects created in the current cycle. **G_SURVIVAL** indicates objects that have survived at least one cycle. **G_OLD0** marks objects classified as old in this cycle. **G_OLD1** represents objects that having completed first full cycle as old. **G_OLD** describes objects that have survived at least two full cycles as old. **G_TOUCHED1** refers to old objects with references to new objects in the current cycle, while **G_TOUCHED2** applies to old objects with references to new objects from the previous cycle. A key invariant is that truly old objects cannot reference new ones.

2.4 Work and Debt

In incremental and generational garbage collectors, work is divided into manageable steps, measured by metrics such as bytes scanned, objects marked, or finalizers invoked (e.g., scanning 1 byte is 1 unit of work, invoking a finalizer is 50 units).

Debt, representing the remaining work, adjusts with object allocation and collection. It increases with allocation and decreases with collection, and the system triggers the next cycle when debt reaches a threshold. This mechanism maintains efficient workload management and balance in the garbage collection process. Details on debt management for specific collectors will follow.

2.5 Freeing an Object

freeobj handles the deallocation of unreachable objects. **luaH_free** deallocates Lua tables by freeing the hash part with **freehash**, the array part with **luaM_freearray**, and the table structure with **luaM_free**.

luaM_free is a macro that calls **luaM_free_**, which performs the actual memory deallocation via **l_alloc** using standard C library functions.

3 Stages of a Garbage Collection Cycle

Understanding the stages of a garbage collection cycle is essential before exploring the different garbage collectors in Lua. Although some stages are common to all collectors, others differ based on the specific type of collector.

The Lua garbage collection cycle involves nine stages: **GCSpropagate**, **GCSenteratomic**, **GCSatomic**, **GCSswpallgc**, **GCSswpfinobj**, **GCSswptobefnz**, **GCSswpend**, **GCScallfin**, and **GCSpause**. The cycle starts and ends in the **GCSpause** state, with the **singlestep** function managing the execution of each stage and transitioning between states. Details of each stage's tasks are outlined below.

3.1 GCSpause

The collection process begins in the **GCSpause** state, where the **restartcollection** function is invoked. This function clears the gray list and marks the root set, global metatables, and registry. It also marks any objects that were being finalized in the previous cycle but whose finalizers have not yet been executed. After completing these tasks, the state transitions to **GCSpropagate**.

3.2 GCSpropagate

In the **GCSpropagate** state, the `gray` list is traversed until empty. The **propagatemark** function is responsible for marking various object types. Objects that require further processing in the atomic phase are added to the `grayagain` list. When no more gray objects remain, the state transitions to **GCSenteratomic**.

3.3 GCSenteratomic

The **GCSenteratomic** function calls the **atomic** function, which changes the state to **GCSatomic**. This is the entry point for the atomic phase through the **singlestep** function.

3.4 GCSatomic

The **GCSatomic** phase is crucial in the garbage collection cycle, especially for incremental and generational collectors, and must complete in a single pause. It involves several key steps:

First, the current `grayagain` list [2.2] is saved in a temporary variable, and `grayagain` is set to `NULL`. The running thread, global metatables, and registry are then marked. Next, the `gray` [2.2] and `grayagain` lists are traversed to mark all reachable objects.

The **convergeephemerons** function is called to process `ephemeron tables` [B], propagating marks from keys to values until no more marks can be propagated, with alternating traversal directions to improve efficiency. `Weak tables` [B] are then updated to clear values with dead keys, and the status of these tables is recorded.

Objects with finalizers marked for collection are moved to the `tobefnz` list and re-marked due to potential resurrection [A.1]. The **propagateall** function is invoked to propagate changes from re-marking, followed by another call to **convergeephemerons** for additional propagation in ephemeron tables.

Finally, the API string cache is cleared, and the current white is switched to the other white (from `WHITE0` to `WHITE1` or vice versa) to distinguish between objects marked in this step and those in the subsequent marking step.

At the end of this phase, the **entersweep** function is called, changing the state to **GCSswpallgc**. This function updates the object list pointer to refer to an object within the list (bypassing the header), skips dead objects using **sweeptolive**, and estimates the total memory allocated.

Notably, this is the only phase where the invariant of no black object referencing a white object [2.2] is violated.

sweepstep Function

The **sweepstep** function is vital in the garbage collection process. It utilizes the **sweeplist** function to process a list of objects. During this phase:

First, each object is checked; if its color is non-current `white` (indicating it is dead), it is freed using the **freeobj** [2.5] function. Objects that are not dead have their color updated to current white, preparing them for the next cycle. The function removes up to `GCSWEEPMAX` objects from the list and returns a pointer to the next object for future processing. Additionally, the total memory allocated is adjusted by subtracting the size of the freed objects.

Following this, the **sweepstep** function updates the object list pointer to the next list to be processed. If the object list pointer is NULL, the state is changed to **GCSswpend**.

3.5 GCSswpallgc State

In the **GCSswpallgc** state, the **sweepstep** function [3.4] is called to sweep all regular objects. This process involves sweeping the list of all regular objects, updating their colors, and freeing dead objects. Once the list is fully processed, the state transitions to **GCSswpfinobj**.

3.6 GCSswpfinobj State

During the **GCSswpfinobj** state, the **sweepstep** function [3.4] processes the list of all objects with finalizers. When the end of this list is reached, the state transitions to **GCSswptobefnz**.

3.7 GCSswptobefnz State

In the **GCSswptobefnz** state, the **sweepstep** function [3.4] sweeps the list of objects marked for finalization (Objects with finalizer marked as dead). Once this list is fully processed, the state transitions to **GCSswpend**.

3.8 GCSswpend State

When in the `GCSswpend` state, the **checkSizes** function is invoked to assess the sizes of the string table and attempt to shrink it if possible. Subsequently, the state transitions to **GCScallfin**.

3.9 GCScallfin State

In the **GCScallfin** state, the **runafewfinalizers** function executes a limited number of finalizers (up to 10). This approach prevents excessive delays that could result from running all finalizers in one step. Once this process is completed, the state changes to **GCSpause** marking the end of the garbage collection cycle.

4 Stop-the-World Garbage Collection

In Lua, a stop-the-world garbage collection cycle can be initiated by calling the **luaC_fullgc** function. From the user space, this can be triggered by first turning off the default incremental mode using **lua_gc(L, LUA_GCSTOP, 0)**(`lapi.c`) which sets `gcstp` to `GCSTPUSR` and then calling **lua_gc(L, LUA_GCRESTART, 0)**(`lapi.c`) which clears any remaining cycles and sets `gcstp` to 0 and sets `GCdebt` to 0 by calling **luaE_setdebt**(`lstate.c`) function. Finally, the **lua_gc(L, LUA_GCCOLLECT, 0)**(`lapi.c`) function call will start the garbage collection cycle.

The **luaC_fullgc** function executes all the stages of a garbage collection cycle in a single pause by invoking the **fullinc** function. The flow of the function calls are as follows as found in the graph (some functions from source code are not visible possibly due to function inlining or less instructions spent)

lua_gc(L, LUA_GCCOLLECT, 0) → **luaC_fullgc** → **singlestep** → **entersweep** → **sweeplist** → **freeobj** → **luaH_free** → **luaM_free** → **l_alloc**

4.1 Execution Flow

1. **lua_gc(L, LUA_GCCOLLECT, 0)** triggers the garbage collection cycle.
2. **luaC_fullgc** executes all stages of the garbage collection cycle in a single pause. It calls the **fullinc** function to manage incremental steps and uses the **luaC_runtilstate** function to transition between states with the **singlestep** [3] function.
3. **singlestep** completes any remaining steps from the previous cycle before starting a new one. If any marked objects are found, they are processed in the **entersweep** function to mark them back to white, and the state is reset to **GCSpause**.

4. The **Marking Phase** begins. In the **GCSpause** state, the root set, global metatables, and registry are marked. The **GCSpropagate** state is skipped as traversal will eventually be completed in the **GCSatomic** state.
5. **singlestep** advances the garbage collection cycle step-by-step, transitioning between states. In the atomic phase, the gray list is traversed, and objects are marked as black. The **convergeephemerons** function processes ephemeron tables, and weak tables are updated to clear dead keys.
6. The **Sweeping Phase** begins with the **GCSswpallgc** state, followed by **GCSswpfinobj** and **GCSswptobefnz**. The **sweepstep** function [3.4] is called to sweep through lists of objects, freeing dead objects and updating memory allocation. Object traversal follows the steps in the **sweepstep** function, first for regular objects and then for objects with finalizers.
7. The **freeobj** [2.5] function deallocates memory using **luaH_free** for tables and **luaM_free** for other objects. The **luaM_free** function uses **l_alloc** to deallocate memory with standard C library functions.

Essentially, the stop-the-world garbage collection cycle in Lua complets all stages in a single pause, ensuring that all objects are marked, swept, and finalized before the cycle ends as discussed in section [3].

During a stop-the-world garbage collection cycle, the entire program is paused until the collection completes. This pause can be noticeable in programs with large heaps or heavy load, leading to potential interruptions in responsiveness, which is problematic for real-time applications requiring low latency.

5 Incremental Garbage Collection

In Lua, the incremental garbage collector is designed to minimize the impact of garbage collection pauses on program execution. Instead of performing the entire garbage collection cycle in one go, the incremental collector interleaves the collection process with program execution. This approach allows the garbage collector to perform small, incremental steps reducing the duration of each pause.

5.1 Parameters

Three parameters control the behavior of the incremental garbage collector:

5.1.1 Pause

The garbage collector's pause setting controls the delay before starting a new cycle. The collector begins a new cycle when memory usage reaches `n%` of the usage after the last collection. Larger values result in a less aggressive collector. A value of 200 means the collector waits for memory usage to double before initiating a new cycle. The default value is 200, with a maximum of 1000.

5.1.2 Step multiplier

The garbage-collector step multiplier controls the speed of the collector relative to memory allocation by determining how many elements are marked or swept for each kilobyte of memory allocated. Larger values make the collector more aggressive, increasing the size of each incremental step. It is recommended to avoid values below 100, as they may cause the collector to become too slow and potentially prevent it from completing a cycle. The default value is 100, and the maximum allowable value is 1000. Essntailly, it converts debt [2.4] from bytes to work units as

$$\text{debt} = \frac{g \to \text{GCdebt}}{\text{WORK2MEM}} \times \text{stepmul} \tag{1}$$

Where WORK2MEM is a macro that converts gives size of Tagged Value (Basic representation of values in Lua: an actual value plus a tag with its type.)

5.1.3 Step size

The garbage-collector step size controls the amount of memory allocated by the interpreter before performing a garbage collection step. This parameter is logarithmic: a value of n means the interpreter will allocate 2^n bytes between two consecutive steps of a single cycle. Large value, such as 60, turns the collector into a stop-the-world (non-incremental) collector. The default value is 13, which results in steps of approximately 8 KB.

5.2 Incremental Garbage Collection Cycle

5.2.1 Initializing Incremental Mode

- **lua_gc(L, LUA_GCINC, LUAI_GCPAUSE, LUAI_GCMUL, LUAI_GCSTEPSIZE)**: This function initializes the incremental garbage collection (GC) mode. Here, **LUAI_GCPAUSE**, **LUAI_GCMUL**, and **LUAI_GCSTEPSIZE** are default values representing the pause, step multiplier, and step size respectively. This function calls **luaC_changemode(L, KGC_INC)** (`lgc.c`), which sets the GC to incremental mode. This function ultimately calls **luaC_changemode(L, KGC_INC)** (`lgc.c`), which changes the GC mode to incremental.

5.2.2 Preparing for Incremental Mode

- **enterinc(g)**: Prepares the GC for incremental mode by:
  - Marking all objects as white (unmarked). **whitelist** function iterates through lists of objects (like `allgc`, `finobj`, and `tobefnz`), marking all elements as white and resetting their age. This ensures all objects are treated as new, except for fixed strings.
  - Resetting intermediate lists (such as `old1` and `survival`) to `NULL`.
  - Setting the GC state to `GCSpause`, indicating readiness for the next cycle.

5.2.3 Garbage Collection Steps

An incremental garbage collection cycle is triggered based on the `pause` parameter and is divided into multiple steps, which are interleaved with the execution of the program. Each step is executed whenever the interpreter allocates a certain amount of memory, as determined by the step size parameter.

The function **incstep** oversees each step of the incremental garbage collection cycle by invoking the **singlestep** function to manage the tasks associated with each stage. Specifically, **incstep** performs the following actions:

- Converts the `debt` into work units using the `stepmul` value, calculates `stepsize` as 2^{stepsize}, and translates this into the corresponding work units.
- Calls **singlestep** function repeatedly to perform garbage collection steps until the debt is less than the `stepsize`. The transitions from one stage to the next are managed by **singlestep** itself. The job of **incstep** is to ensure that the garbage collection cycle is executed incrementally. The stages of the incremental garbage collection cycle are same as described in section [3]. The only difference is that these stages are executed in multiple steps rather than all at once allowing the program to continue executing between steps.

- Updates the debt to reflect the number of bytes remaining to be processed and sets this updated value as the new debt for the next step by converting it back to bytes.

Important considerations for incremental garbage collection:

- All stages except **GCSatomic** can be executed across multiple steps. For instance, the **GCSpropagate** stage involves traversing the gray list, which can be done in multiple steps rather than all at once.
- The **GCSatomic** stags must be completed in a single step to ensure the atomic phase is handled during a single pause. Consequently, the steps involving these stages are expected to be longer in duration.

6 Generational Garbage Collection

The generational garbage collector is based on the hypothesis that *most objects die young* and so the collector can focus on the young generation of objects. The age-based marking scheme [2.3] is used to segregate objects into different generations based on their age.

6.1 Parameters

Two parameters control the behavior of the generational garbage collector:

6.1.1 Minor multiplier

The minor multiplier controls the frequency of minor collections. For a minor multiplier x, a new minor collection will be done when memory grows x% larger than the memory in use after the previous major collection. For instance, for a multiplier of 20, the collector will do a minor collection when the use of memory gets 20% larger than the use after the previous major collection. The default value is 20; the maximum value is 200.

After collcetions in generational mode the debt for the next cycle is calculated by **setminordebt** as

$$\text{debt} = -\left(\frac{\text{gettotalbytes}(g)}{100}\right) \times \text{g} \rightarrow \text{genminormul} \tag{2}$$

where gettotalbytes is a macro that returns the total memory allocated.

6.1.2 Major multiplier

The major multiplier controls the frequency of major collections. For a major multiplier x, a new major collection will be done when memory grows x% larger than the memory in use after the previous major collection. For instance, for a multiplier of 100, the collector will do a major collection when the use of memory gets larger than twice the use after the previous collection. The default value is 100; the maximum value is 1000.

6.2 Generational Garbage Collection Flow

6.2.1 Initialization

The generational garbage collector can be initiated as follows:

1. **lua_gc(L, LUA_GCGEN, LUAI_GENMINORMUL, LUAI_GENMAJORMUL)** (`lapi.c`): Initializes the generational GC mode with `LUAI_GENMINORMUL` and `LUAI_GENMAJORMUL` representing the minor and major multipliers, respectively. This function calls **luaC_changemode(L, KGC_GEN)** (`lgc.c`), which sets the GC mode to generational by invoking the **entergen** function (`lgc.c`).

6.2.2 Preparatory Steps

1. **entergen(L, g):** Prepares the system for generational mode by completing an atomic cycle and transitioning all objects to the `old` state. The specific steps include:
   (a) **luaC_runtilstate(L, bitmask(GCSpause))**: Runs the GC until it reaches the pause state, stopping ongoing collections and stabilizing the heap.
   (b) **luaC_runtilstate(L, bitmask(GCSpropagate))**: Begins a new GC cycle by entering the propagation phase, marking live objects, and setting up for subsequent collection.
   (c) **atomic(L):** Executes the atomic phase, marking strongly reachable objects, handling resurrected objects, and removing dead ones.
   (d) **atomic2gen(L, g):** Clears gray lists, sweeps dead objects, and marks surviving objects as `old` Threads are placed on the `grayagain` list, and other objects are fully marked.
   (e) **cleargraylists(g):** Clears gray lists to prevent further marking and facilitate the transition to generational mode.
   (f) **sweep2old(L, &g->lists):** Sweeps through specified object lists, marking survivors as "old" for less frequent collection.
   (g) **setminordebt(g):** Sets the debt for the next minor GC cycle to focus on newer objects.

6.2.3 Execution

The **genstep** function handles generational garbage collection:

1. If the previous collection was ineffective, **genstep** performs a full major collection using **stepgenfull**.
2. If memory usage has increased significantly since the last major collection, it triggers a major collection with **fullgen**.
3. If this major collection reclaims a significant amount of memory, minor collections continue; otherwise, the collection is marked as `bad`, and the next major collection is delayed using **setpause**.
4. Typically, **genstep** performs a minor collection with **youngcollection** and updates the debt with **setminordebt**.

The details of the minor and major collection steps are as follows:

6.2.4 Young Collection

This step is executed if the amount of memory allocated since the last collection is greater than the minor multiplier but less than the major multiplier. Before we go ahead with the exact steps, let's understand some of the functions used and their purpose:

- **sweepgen**: Responsible for the sweeping phase in the generational mode. It deletes white non-old objects, promotes objects from `G_NEW` to `G_SURVIVAL`, resets their color from `G_OLD0` to `G_OLD1`, and from `G_OLD1` to `G_OLD`. `TOUCHED` objects are not promoted here.
- **correctgraylist**: Corrects a list of gray objects by removing non-gray objects and promoting `G_TOUCHED1` objects to `G_TOUCHED2` and `G_TOUCHED2` to `G_OLD`, changing their color to black.

Steps of the young collection:

- Marks all items in the `OLD1` list as `OLD`.
- Performs an atomic step and mark all objects in the `gray` list.

- Calls the **sweepgen** function for regular objects, excluding those older than or as old as `G_OLD1`. Then calls **sweepgen** for objects with finalizers.
- Calls the **finishgencycle** function to complete the young collection, which in turn calls **correctgraylist** for the `grayagain` list and `weak tables` [B].

6.2.5 Full Generation Collection

This step is executed if the amount of memory allocated since the last collection exceeds the major multiplier. It invokes the **fullgen** function, which performs a full garbage collection cycle by temporarily switching to incremental mode and then back to generational mode. After this step, the collector checks if the amount of memory freed is at least half of the memory allocated since the last collection. If so, the garbage collector remains in generational mode and schedules the next cycle as a minor cycle. Otherwise, this collection is deemed a `bad collection` and the **stepgenfull** function is invoked.

stepgenfull: The `stepgenfull` function determines whether the garbage collector should stay in incremental mode or switch back to generational mode after a `bad` collection:

- Checks if the collector is still in generational mode. If so, it switches to incremental mode.
- Begins a new garbage collection cycle and counts the number of objects traversed.
- If fewer than 9/8 of the objects compared to the last collection are traversed, the collection is considered `good` and the collector returns to generational mode.
- If the collection traverses more objects, the collector stays in incremental mode to avoid ineffective minor collections.
- Finally, the funtcion computes the estimeate for total memory allocated.

References

[1] Garbage Collection in Lua, Roberto Ierusalimschy, PUC-Rio: LiM'19 talk - *YouTube*

[2] Garbage Collection in Lua, Roberto Ierusalimschy, PUC-Rio: LiM'19 slides - *Lua Workshop 2018*

[3] Lua Source Code - *GitHub*

[4] Lua 5.4 Reference Manual - *Lua.org*

Appendices

A Finalizers

Lua provides a mechanism called finalizers to run a function when an object is about to be collected. Finalizers are particularly useful for releasing resources and performing cleanup before an object is destroyed. When userdata is about to be collected and its metatable has a `__gc` field, Lua calls the function stored in this field, passing the userdata as an argument to perform necessary cleanup operations.

A.1 Resurrection

From the perspective of a garbage collector, an object marked as unreachable is generally considered eligible for collection. However, if the object has a finalizer, the object, along with any other objects accessible through it, must be preserved until the finalizer has been executed. In other words, the garbage collector cannot free such objects until they are truly dead and their finalizer has been run.

B Weak Tables

Lua offers a special table type called weak tables, where elements don't prevent garbage collection. A table's "weakness" is defined by its mode: weak keys, weak values, or both.

An ephemeron table has weak keys and strong values, where a value is reachable only if its key is reachable. If a key is referenced only by its value, the key-value pair is removed.

B.1 Weak Tables and Resurrected Objects

Resurrected objects, either being finalized or accessible through others, are removed from weak values before finalizers to prevent unintended resurrection. These objects remain in weak keys until the next collection cycle, allowing finalizers to access them via the weak table.

C Barrier

A barrier is used to maintain the invariants of tri-color marking and age-based marking. Barriers are triggered when a black object references a white object. There are two types of barriers:

C.1 Forward Barrier

The forward barrier marks the white object as gray to ensure it is revisited by the garbage collector. In generational garbage collection, if the black object is old, the white object is marked as `OLD0`, and may transition to `OLD1` and then `OLD` in subsequent cycles.

C.2 Backward Barrier

The backward barrier marks the black object as gray again to ensure it will be revisited by the garbage collector. In generational mode, if the black object is old and has the age `G_TOUCHED2`, it is marked to transition to `G_TOUCHED1`. If the black object is not yet gray, it is added to the `grayagain` list and marked gray for processing in the next cycle.

The choice of wh