Optimizing Lua code

This optimizing guide is targeted at *LuaJIT 2.1.0-beta3* with the JIT compiler disabled.

Table of contents

The Golden Rule – Profile before and after optimizing

Programmer time is just as valuable as execution time. The key is striking a good balance between development time, code clarity, flexibility and performance. Work smarter, not harder.

> “Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97% of the time: _____ _____. Yet we should not pass up our opportunities in that critical 3%. **– Donald Knuth**
> ”

Measuring techniques

Microprofiler

Fence blocks with profiling code to make sure you're optimizing the right thing **and** that your changes actually have beneficial impact.

```
Profiler.start(scope_name)
--[[ code to profile goes here ]]
Profiler.stop(scope_name)
```

Bytecode inspection

Generated LuaJIT code can be inspected using `luajit -bl`. Code doesn't need to execute, just be syntactically correct so it can be parsed.

```
$ luajit -bl - <<EOF
local a = {}
a:test():test()
EOF
-- BYTECODE -- stdin:0-3
0001    TNEW     0   0
0002    MOV      2   0
0003    TGETS    1   0   0  ; "test"
0004    CALL     1   2   2
0005    MOV      2   1
0006    TGETS    1   1   0  ; "test"
0007    CALL     1   1   2
0008    RET0     0   1
```

Microbenchmarking

A microbenchmark is a program to measure and test the performance of a very small piece of code. In general, programming decisions should *not* be solely taken based on the result of a microbenchmark. However, they can be useful to guide a programmer to take a decision if there aren't any other considerations.

See the following document for some microbenchmark results. Do NOT take results as gospel, make sure to test they apply in your case.

Unknown macro: 'lref-gdrive-file'

Optimization guidelines

Specify the shape of tables at creation.

When the shape of a table is known at compile time, Lua generates bytecode that allocates necessary space and avoiding some rehashing. The engine also provides some functions to create tables with space pre-allocated.

DO

```
obj = {
    _current_value = nil,
    set_value = function(self, val)
        self._current_value = val
    end,
}

list = { "one", "two", "three" } -- Initial array part is already scaled to fit 3 elements.

array = Script.new_array(3) -- Reserve three entries for an array structure

map = Script.new_map(3) -- Reserve three slots for key/value pairs

t = Script.new_table(2, 3) -- Reserve two entries for an array structure AND 3 slots for key/value pairs
```

DON'T

```
obj = {}
obj._current_value = nil
function obj:set_value(val)
    self._current_value = val
end

list = {}
list[1] = "one" -- Array part resize.
list[2] = "two"
list[3] = "three" -- Array part resize.
```

Avoid creating new tables except for during setup.

Table allocation and garbage collection are both expensive. There are several techniques to avoid table creation, useful in different situations

A common idiom is to pass a table in which to put in the results. That way the caller can control when the allocation happens (once at startup, at init, every call, etc.).

DO

```
local function my_function(t)
    t[1] = "foo"
    t[2] = "bar"
    t[3] = "baz"
    return t, 3 -- Returns the number of inserted elements, so the caller doesn't need to use the # operator.
end
```

If you're short on time and optimizing existing code and maybe a full rewrite isn't an option, you can use table.clear() or table.clear_array() on an existing table. It's more expensive than the first option, but cheaper than creating a new table

DO

```
local function my_function(t)
```

```
    table.clear_array(t) -- Clears the array by looping over all keys and setting them to nil
    t[1] = "foo"
    t[2] = "bar"
    t[3] = "baz"
    return t
end
```

DON'T

```
local function my_function(t)
    local t = {} -- Allocation.
    t[1] = "foo"
    t[2] = "bar"
    t[3] = "baz"
    return t
end
```

Use a local to avoid repeated global/table lookups.

Table lookups are cheap, but accessing a local is even cheaper. This can add up if the same function is called many times.

NOTE: Considerations are a bit different when the JIT compiler is enabled, due to aliasing concerns.

DO

```
local time_manager = Managers.time
local t_game = time_manager:time("game")
local t_main = time_manager:time("main")
print(t_game - t_main)

local print, tostring = print, tostring -- Converts globals into locals.
for k, v in pairs(_G) do
    print(tostring(k), tostring(v)
end
```

DON'T

```
local t_game = Managers.time:time("game")
local t_main = Managers.time:time("main")
print(t_game - t_main)

for k, v in pairs(_G) do
    print(tostring(k), tostring(v)) -- Does 3 global lookups each iteration.
end
```

Prefer numeric for-loops over ipairs to iterate over arrays.

The `ipairs` idiom incurs the overhead of a function call each frame, and checks the validity of its arguments every call. The numeric form is tighter and it's easy to iterate backwards, so elements can be removed during the loop.

NOTE: An `pairs` loop *can* be faster than a numeric for-loop in specific conditions, as it's handled in a special way. But a base for-loop is always faster than `ipairs`.

DO

```
for i=1, #list do
    print(list[i])
end
```

DON'T

```
for i, v in ipairs(list) do
    print(v)
```

```
end
```

Take advantage of BC specialization.

LuaJIT can emit type-specialized bytecode for arithmetic ops with constants, and table accesses using a constant key.

DO

```
local base_critical_strike_chance = CareerSetting["dr_ironbreaker"] -- Uses TGETS (specialized to string key)
    .attributes
    .base_critical_strike_chance

local crit_percent = 100 * base_critical_strike_chance -- Uses MULNV (specialized to multiplying by the LHS b
```

DON'T

```
local career_name = "dr_ironbreaker"
local crit_chance = CareerSettings[career_name] -- Uses TGETV.
    .attributes
    .base_critical_strike_chance

local CHANCE_TO_PERCENT = 100
local crit_percent = CHANCE_TO_PERCENT * crit_chance -- Uses MULVV.
```

Avoid creating too many single-use locals.

Whenever a local is defined, LuaJIT allocates 8 bytes of stack space to store it's value. Additionally, the containing function also needs to store the lines it's defined in and it's name as debug data.

DO

```
-- Spreading the accesses across multiple lines allows us to get error information.
this._button_widget
    .content
    .button_hotspot
    .disable_button = true
```

DON'T

```
local button_widget = this._button_widget
local content = button_widget.content
local button_hotspot = content.button_hotspot
button_hotspot.disable_button = true
```

When writing if-else or if-elseif, put the most common occurring case(s) first.

There are two main advantages:

- Less checks are needed to reach the most common case.
- If the most common branch is first, it's probably already partially loaded into cache.

DO

```
if state == "sky_is_blue" then
    --[[ ... ]]
elseif state == "stars_are_aligned" then
    --[[ ... ]]
end
```

DON'T

```
if state == "stars_are_aligned" then
    --[[ ... ]]
elseif state == "sky_is_blue" then
    --[[ ... ]]
end
```

When possible, use tail-calls.

A (proper) tail call is when a function returns the results of calling another function (including itself). Tail calls are optimized to re-use the same stack frame as the currently executing function.

DO

```
function MyClass:method()
    return this._some_handler:get_x()
end
```

DON'T

```
function MyClass:method()
    local x = this._some_handler:get_x()
    return x
end
```

Avoid the unary length (#) operator on tables.

The length of a table is defined as ANY natural number N such that `t[N]~=nil` and `t[N+1]==nil`; or 0 if there is no such N. The length operator does binary search to find the N and is a O(log N) operation. Since it's so common it's optimized quite a bit but to squeeze performance you should try to avoid it if possible.

NOTE: The length operator on strings is O(1) because the length is stored internally (strings are interned).

DO

```
local t, i = {}, 0
for k, v in pairs(_G) do
    i = i + 1
    t[i] = k
end
```

DON'T

```
local t = {}
for k, v in pairs(_G) do
    t[#t+1] = k
end
```

String creation is expensive; string reads is cheap

Reading the length of a string is O(1). Strings are interned so string comparison is just pointer comparison and thus O(1). Creating new strings has a cost. Avoid creating strings during a frame.

DO

```
-- Cache strings instead of concatenating every frame.
self._key = "prefix_"..some_value

-- Use string comparisons
if value == "red" then
```

DON'T

```
-- Concatenating strings every frame.
local v = tab["prefix_"..some_value]

-- (Over)use enum structures.
if value == ColorEnum.red then
```

Avoid engine function calls

Crossing the language boundary between C and Lua has some overhead. Try to keep code contained in either without mixing them too much whenever it is possible. Note that built-in Lua functions are fast.

DO

```
local x = math.random() -- Lua built-in: fast and high quality.

PLATFORM = Application.platform() -- Do once at startup.
--[[ ... ]]

local platform = PLATFORM -- Access the global.
```

DON'T

```
local x = Math.random() -- Engine API: slow *and* bad.

-- Calling an engine function.
local platform = Application.platform()
```

Further reading

- https://www.lua.org/gems/sample.pdf, Roberto Ierusalimschy (for PUC-Lua, but has sections applicable to LuaJIT too).
- http://bitsquid.blogspot.com/2011/12/pragmatic-approach-to-performance.html, Niklas Frykholm (co-creator of Bitsquid/Stingray).
- http://lua-users.org/lists/lua-l/2009-11/msg00089.html
- https://luajit.org/git/luajit-2.0.git
- http://wiki.luajit.org/Bytecode-2.0
- http://wiki.luajit.org/New-Garbage-Collector
- https://www.reddit.com/user/mikemike/
- http://lambda-the-ultimate.org/node/3851
- https://mrale.ph/talks/vmss16/