Effil: yet another way for multithreading in Lua …

by Mikhail Kupriyanov

Telegram/Twitter: @mihacooper
LinkedIn: mikhail-kupriyanov

Multithreading in Lua

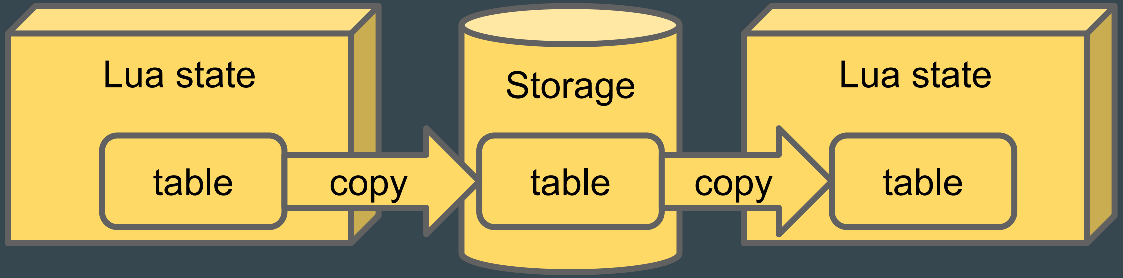
1. Custom Lua interpreter:
   LuaThread

2. State per thread (no data exchange):
   lua-llthreads

3. State per thread + Copying based message passing:
   lua lanes, lua-zmq

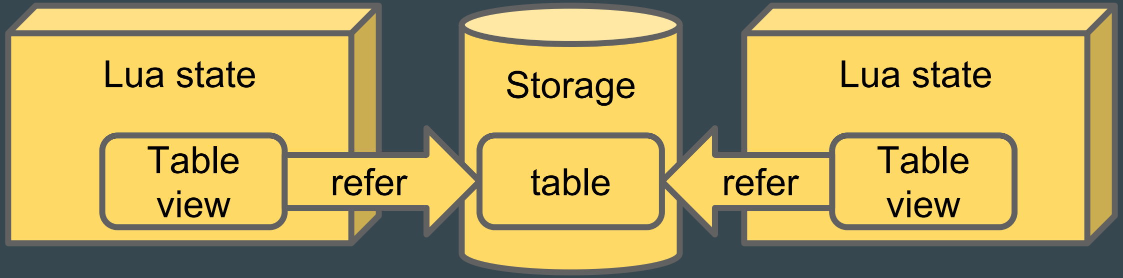
http://lua-users.org/wiki/MultiTasking

Data copying



1. One data exchange = 2 copying operation
2. Not possible to share the same data
3. Hard to support complex data structures

Shared objects



1. Parallel access to the same data
2. No data copying
3. Easy to support complex data types

Shared objects in Effil

1. Shared objects in Effil:
   `effil.table, effil.channel, effil.thread, effil.function`

2. Nested structure of shared objects

3. Recursive references

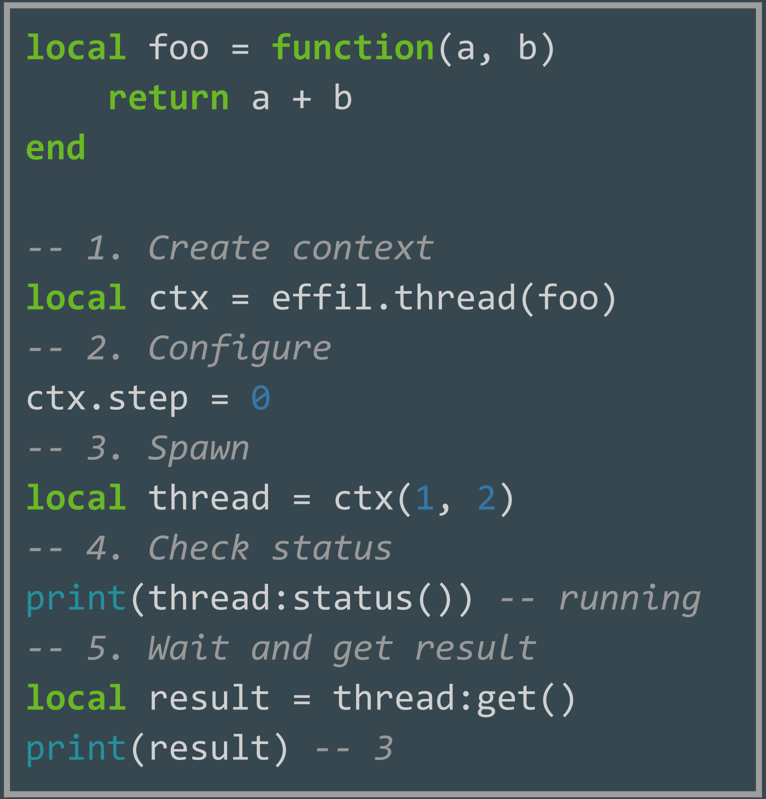
4. Automatic memory management outside of Lua states

effil.gc - Garbage Collector

```
-- Collect garbage
effil.gc.collect()
-- Get amount of allocated objects
effil.gc.count()
-- Set/Get step of GC iteration
effil.gc.step(100)
-- Pause/Resume garbage collecting
effil.gc.pause()
effil.gc.resume()
```

1. Tracing Garbage Collector (tri-color marking)
2. Control lifetime of all shared objects

effil.thread



```lua
local foo = function(a, b)
    return a + b
end

-- 1. Create context
local ctx = effil.thread(foo)
-- 2. Configure
ctx.step = 0
-- 3. Spawn
local thread = ctx(1, 2)
-- 4. Check status
print(thread:status()) -- running
-- 5. Wait and get result
local result = thread:get()
print(result) -- 3
```

1. Thread is shared object
2. Threads are optionally manageable: cancel, pause, resume
3. Thread has status
4. Thread saves stacktrace
5. Helper functions:
    a. `effil.thread_id()`
    b. `effil.yield()`
    c. `effil.sleep()`

effil.channel

```lua
-- 1. Channel with limited capacity
local channel = effil.channel(2)

-- 2. one push creates one message
channel:push(1, 2)
channel:push("hello")

-- 3. Infinitely wait and pop
channel:pop() -- 1  2
-- 4. Wait for 5 seconds and pop
channel:pop(5, "s") -- hello
```

1. Channel is shared object
2. Channel is FIFO message queue
3. Optionally limited capacity
4. Unlimited size of message

effil.table

```
local t = effil.table()
t.key = "value"
t[1] = 1
t[2] = 2
for _, i in ipairs(t) do
    print(i)
end

t.value = 10
effil.setmetatable(t, {
    __add = function(t, v)
        return t.value + v
    end
})
print(t + 20) -- 30
```

1. Table is shared object
2. Can be constructed from Lua table
3. Has all default methods:
   pairs, ipairs, length operator, tostring
4. Supports metatables:
   - `effil.getmetatable`, `effil.setmetatable`
   - `effil.rawget`, `effil.rawset`
   - Standard metamethods
5. Supports recursive tables
6. Persistent global table:
   - `effil.G`

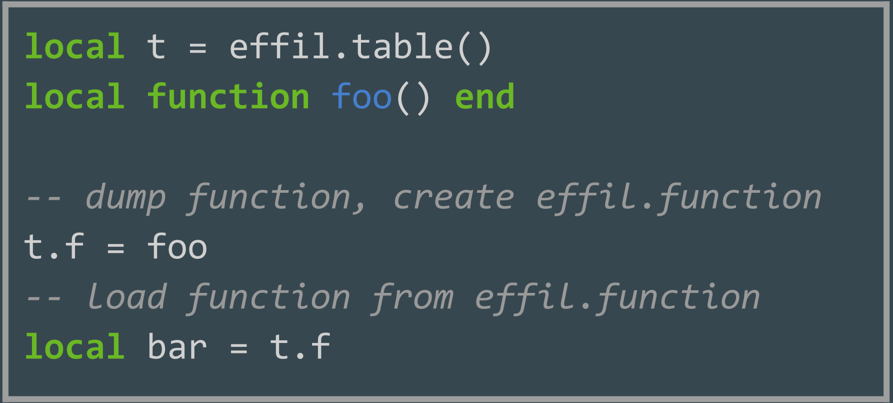
effil.table

```lua
local storage = effil.table({ key = "prefix"})

function worker(t)
    -- t is effil.table here
    storage.key = storage.key .. t.key
end

effil.thread(worker)({key = "_suffix"}):wait()
print(storage.key) -- prefix_suffix
```

Supported Types

1. Primitive types passed by copy:
   - number, string, boolean, nil
2. Tables becomes **effil.table**
3. Functions becomes **effil.function**
   - Hidden type which becomes Lua function on access
4. Not supported:
   - Userdata
   - Lua thread (coroutine)

effil.function



```lua
local t = effil.table()
local function foo() end

-- dump function, create effil.function
t.f = foo
-- load function from effil.function
local bar = t.f
```

1. Function is shared object
2. Function is hidden type
3. Function consist of dumped Lua function and upvalues

Upvalues Problem

```
a = 42 -- global
local t = { 43 }
function foo()
    return a + t[1]
end
```

Lua > 5.1

```
debug.getupvalue(foo, 1) -- _ENV  table: 0x19299f0
debug.getupvalue(foo, 2) -- t  table: 0x19317f0
```

luac -s

```
debug.getupvalue(foo, 1) -- table: 0x19299f0
debug.getupvalue(foo, 2) -- table: 0x19317f0
```

Prohibit Lua table upvalues

```
effil.allow_table_upvalues(false)
```

effil.cache

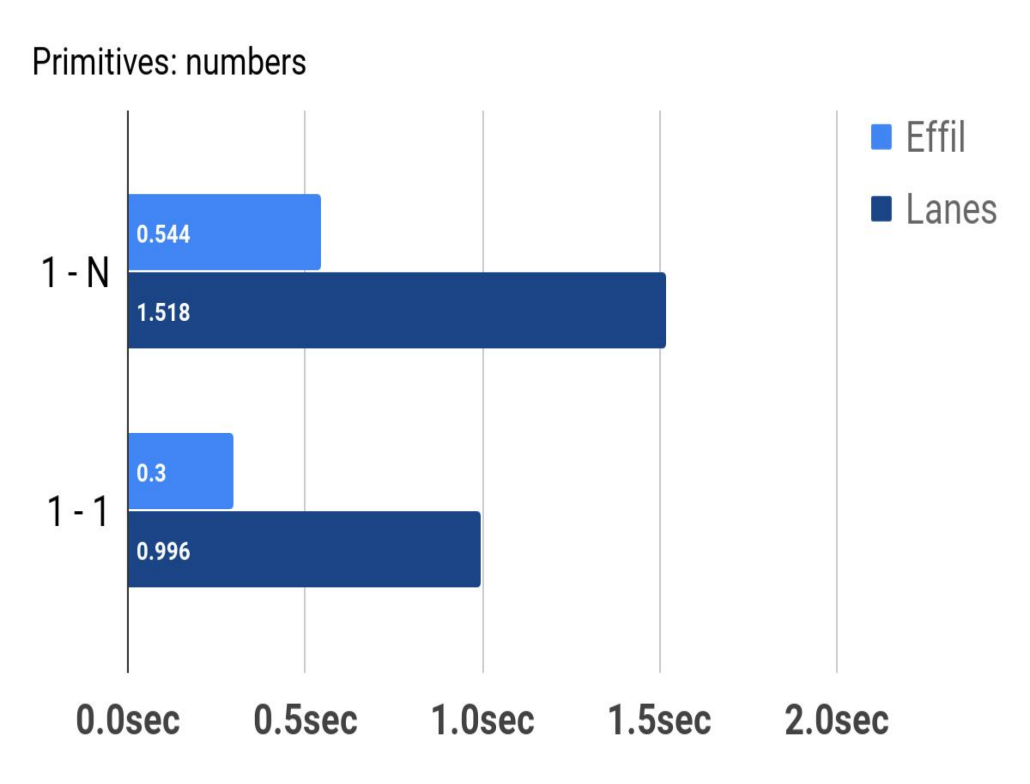
```lua
-- Enable/disable
effil.cache.enabled()
effil.cache.disable()
-- Number of elements in cache
effil.cache.size()
-- Clear cache
effil.cache.clear()
-- Set/get capacity
effil.cache.capacity(10)
-- Remove function from cache
effil.cache.remove(foo)
```

1. Improve performance of effil.function
2. Own cache for each Lua state
3. Two tables:
   - Lua function -> effil.function
   - effil.function -> Lua function
4. Cache is not sensitive to upvalues

Performance Tests

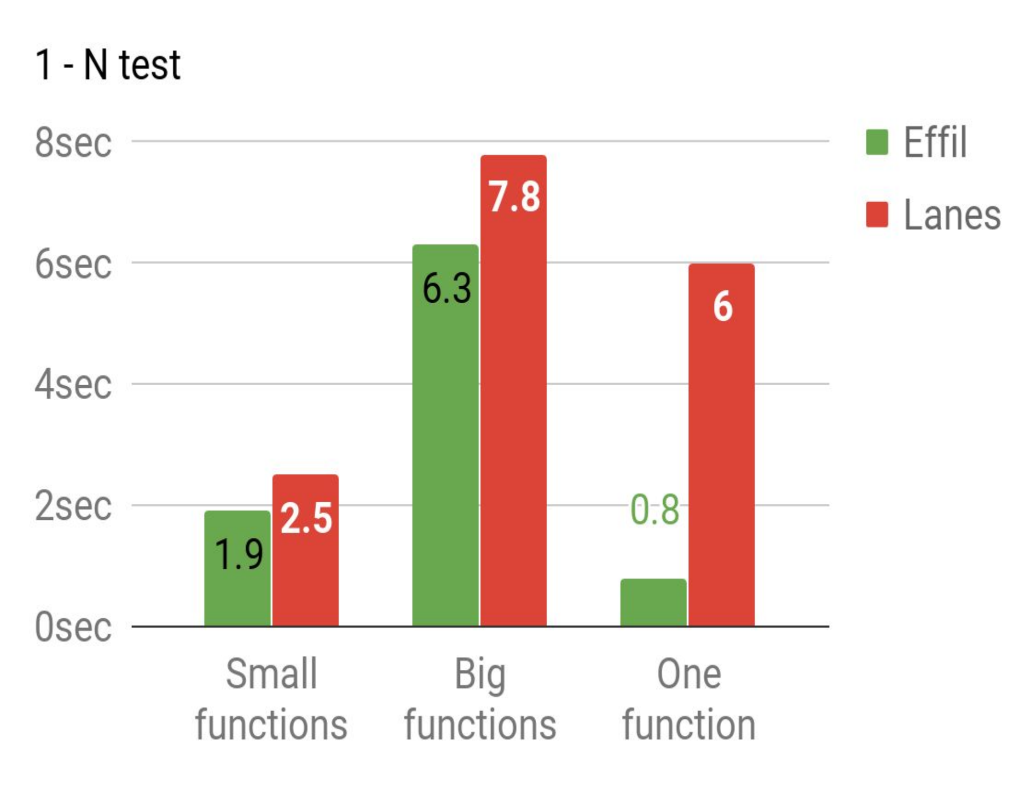
- Compare Effil vs Lanes in message passing
- Configurations:
    - Master thread + workers:
        - One to one (1 - 1)
        - One to many (1 - N), N = 4
    - Two channels: from master to workers and back
    - Transmitted data types:
        - Primitives: strings, numbers
        - Tables:
            - Lua tables, Effil tables (constructed in-place)
            - Small and big tables
            - Unique and repetitive tables
        - Functions
            - Small and Big functions
            - Unique and repetitive functions

Performance Tests: primitives



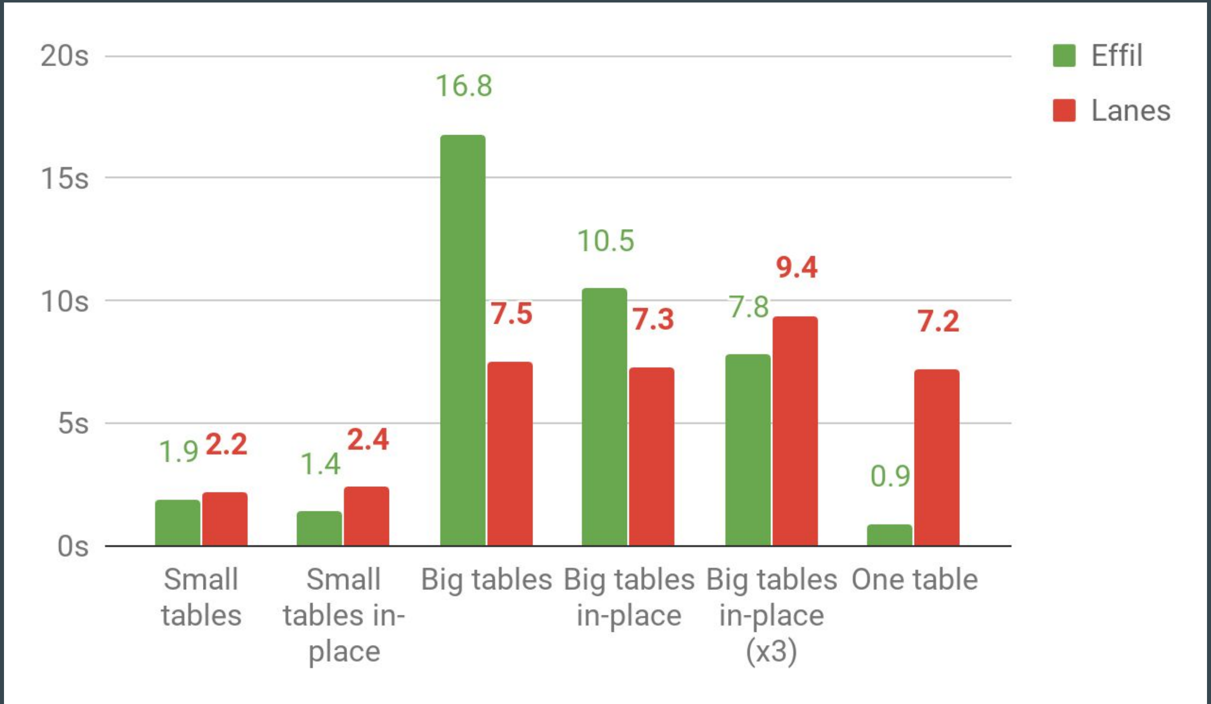
- Results Lanes / Effil:
  - Threads 1 - N: in **2.79** times
  - Threads 1 - 1: in **3.32** times
- Similar results for strings

Performance Tests: functions



- Function dumping/loading shows normal result:
  - Small function: 1 operation
  - Big function: ~500 symbols
- In “one function” effil.cache gets rid of function serialization

Performance Tests: tables



- Serialization is not very fast, use “in-place” approach
- Serialization on small table is normal
- Reuse table if possible

Conclusions

1. High-level multithreading approach with shared objects
2. Good performance especially on reused data
3. Effil supports Linux, MacOS, Windows and LuaJit, 5.1, 5.2, 5.3: https://github.com/effil/effil

Problems:

- Userdata support
- Function environment in upvalues

Further development:

- Synchronization primitives, locking on shared objects, dump effil.table
- Thread pool

Thank you!

https://github.com/effil/effil

Telegram/Twitter: @mihacooper
LinkedIn: mikhail-kupriyanov