RAPTORJIT

a fast, dynamic systems programming language

Max Rottenkolber
max@mr.gy
@eugeneia_
Hello, World!

Hi, I'm Max <max@mr.gy>.

Open source systems hacker

Currently dabbling in high-performance networking applications
Lua

Simple, minimalistic, high-level language, Schemeish semantics, Pascalesque syntax

First class functions, multiple return values, prototype OOP

Central data structure: table (sparse array/hash map hybrid)

Canonical implementation: PUC Lua (simple, embeddable interpreter)
local function hello (name)
    return "Hello, "..name.."!"
end

function greet (name, greeting)
    greeting = greeting or hello
    print(greeting(name or "World"))
end
LuaJIT

Implements a dialect of Lua (5.1½ + goodies)

Strong JIT compiler, efficient implementation (performance competitive with C)

Can express close-to-the-metal programs (native C data access)

Good language for systems programming?
local p = ffi.new('struct {
    uint16_t length;
    char data[10000];
}')

local msg = "Hello, World"

p.length = math.min(#msg, ffi.sizeof(p.data))
ffi.copy(p.data, msg, p.length)
RaptorJIT

Fork of LuaJIT. Goal: to be a really good systems language

Simplify implementation, improve maintainability

Improve JIT for heavy duty server applications (eliminate performance pitfalls, unexpected JIT behavior, provide more reliable performance)

Add features (zero-overhead profiler, introspection tools, + many more to come?)
Simplify and Maintain

🔗 Big bang: Remove all the features that I can live without
#5 by lukego was merged on Mar 12, 2017

Removed support for all architectures except x86_64, Windows, 32-bit heap, ...

Got rid of a TONNE of #ifdefs

Cut code to maintain by ~50%
Simplify and Maintain

LuaJIT interpreter used to be handwritten assembly for each supported architecture.

We almost completed rewriting the interpreter in C (easier to port, easier to change!)

Rationale: we spent 99% of time in compiled code, no use for an overly optimized interpreter.
Simplify and Maintain

lj_str.c: Remove special-case string interning fast-path #150

Merged lukego merged 1 commit into raptorjit:master from lukego:reoptimize-string-intern on Jan 15, 2018
Simplify and Maintain

“Fast-path” bad because:

- tricky custom memcmp routine that needs to be maintained
- slower than "slow-path" (stock memcmp on modern Linux/x86)
- confusing performance behavior: totally unrelated memory allocation could bias an important buffer towards the “fast-path” and impact overall performance
“The fast-path code was written in 2010 and a lot has happened since then [...]. I think the optimization had simply bit-rotted.”
Hacking the JIT

LuaJIT acts as a “best-effort for all use cases” drop-in replacement for PUC Lua. (Fast JIT, when it fails it drops into fast interpreter, huge to solid gains in any case.)

Can we do better for a narrowed use-case?
Hacking the JIT

- interpret
- branch
- record trace
- increment hotcount

hot ➔ branch ➔ record trace
not hot ➔ increment hotcount ➔ branch ➔ record trace
Hacking the JIT

Don't treat the compiler as a black box gifted from a geni(e/ous)!

Study and understand the JIT
Formulate design goals & implement them

⚠️ IAF: Where can a trace end? Why? IAF
#103 opened on Sep 9, 2017 by lukego

⚠️ IAF: What is the difference between a root trace and a side trace? IAF help wanted
question
#99 opened on Sep 5, 2017 by lukego

⚠️ Goal: Avoid "high-impact medium-generality" optimizations goal
#148 opened on Dec 22, 2017 by lukego
Hacking the JIT

LuaJIT aggressively blacklists codepaths that it fails to compile (Good for short running programs, bad for server applications.)

RaptorJIT spends more effort to find traces and provide stable, predictable performance needed for heavy duty server apps.
Hacking the JIT

JIT heuristic updates for stable performance [experimental] #101

Merged  lukego merged 4 commits into  raptorjit:master  from  lukego:long-running-stable  on Dec 11, 2017
Hacking the JIT

LuaJIT doesn't consider the time domain when selecting traces (Causes new traces to be compiled long after initial warmup for code that really isn't hot at all!)

Maybe RaptorJIT should only compile code that is actually executed frequently?

[ draft ] lj_trace.c: clear all hotcounts every second
#260 opened on Oct 9, 2019 by eugeneia
New Features

Replaced LuaJIT profiler with low-overhead, “always on” trace profiler and Auditlog (flight recorder).

Created an interactive tool (Studio) to help understand trace and profile data.
### Inspector on a ByteString (with import <studio>;)}

#### Trace List, Trace Map, VM Profile, Events, Items, Raw, Meta

<table>
<thead>
<tr>
<th>Profile</th>
<th>Samples</th>
<th>Mcode</th>
<th>VM</th>
<th>GC</th>
</tr>
</thead>
<tbody>
<tr>
<td>apps.intel_miptel_mp</td>
<td>5099</td>
<td>99.9%</td>
<td>0.1%</td>
<td>0.0%</td>
</tr>
<tr>
<td>apps.ipv4.arp</td>
<td>910</td>
<td>99.9%</td>
<td>0.1%</td>
<td>0.0%</td>
</tr>
<tr>
<td>apps.ipv4.echo</td>
<td>567</td>
<td>100.0%</td>
<td>0.0%</td>
<td>0.0%</td>
</tr>
<tr>
<td>apps.ipv4.fragment</td>
<td>620</td>
<td>99.8%</td>
<td>0.2%</td>
<td>0.0%</td>
</tr>
<tr>
<td>apps.ipv4.reassemble</td>
<td>37207</td>
<td>12.5%</td>
<td>71.8%</td>
<td>15.7%</td>
</tr>
<tr>
<td>apps.ipv6.arp</td>
<td>640</td>
<td>100.0%</td>
<td>0.0%</td>
<td>0.0%</td>
</tr>
<tr>
<td>apps.ipv6.fragment</td>
<td>1470</td>
<td>100.0%</td>
<td>0.0%</td>
<td>0.0%</td>
</tr>
<tr>
<td>apps.ipv6.reassemble</td>
<td>599</td>
<td>97.7%</td>
<td>2.3%</td>
<td>0.0%</td>
</tr>
<tr>
<td>apps.ipv6.reassemble</td>
<td>11678</td>
<td>97.7%</td>
<td>0.1%</td>
<td>2.2%</td>
</tr>
<tr>
<td>apps.ipv6.reassemble</td>
<td>1029</td>
<td>97.2%</td>
<td>2.3%</td>
<td>0.5%</td>
</tr>
<tr>
<td>engine</td>
<td>1858</td>
<td>79.2%</td>
<td>20.6%</td>
<td>0.2%</td>
</tr>
<tr>
<td>program</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Source code locations of root traces that are hot in the selected profile

<table>
<thead>
<tr>
<th>Location</th>
<th>Samples</th>
<th>Mcode</th>
<th>VM</th>
<th>GC</th>
<th>#Root</th>
<th>#Side</th>
</tr>
</thead>
<tbody>
<tr>
<td>(app)breathe:14 (core/app.lua:591)</td>
<td>32523</td>
<td>0.0%</td>
<td>82.1%</td>
<td>17.9%</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>(link)receive:1 (core/link.lua:48)</td>
<td>1575</td>
<td>99.6%</td>
<td>0.0%</td>
<td>0.4%</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>(link)htons:1 (core/liblua:379)</td>
<td>1108</td>
<td>100.0%</td>
<td>0.0%</td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>(counter)add:1 (core/counter.lua:91)</td>
<td>1091</td>
<td>99.0%</td>
<td>0.3%</td>
<td>0.7%</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>(reassemble)ipv4_packet_has_valid_length1 (appsipv4:670)</td>
<td>18</td>
<td>77.8%</td>
<td>22.0%</td>
<td>0.0%</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>(app)with_restart:1 (core/app.lua:128)</td>
<td>9</td>
<td>100.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>(alarms)Alarmcheck1 (libyang:alarms.lua:591)</td>
<td>8</td>
<td>0.0%</td>
<td>75.0%</td>
<td>25.0%</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>(alarms)Encoderuint32:1 (libptrie:alarms.lua:77)</td>
<td>4</td>
<td>0.0%</td>
<td>100.0%</td>
<td>0.0%</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>(app)now:1 (core/app.lua:121)</td>
<td>1</td>
<td>100.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

#### Root traces starting at selected location (and their side-traces as children)

<table>
<thead>
<tr>
<th>Trace</th>
<th>Samples</th>
<th>Link</th>
<th>Mcode</th>
<th>VM</th>
<th>GC</th>
<th>Start line</th>
<th>Stop line</th>
</tr>
</thead>
<tbody>
<tr>
<td>96</td>
<td>32523</td>
<td></td>
<td></td>
<td>82.1%</td>
<td>17.9%</td>
<td>(app)breathe:14 (core/app.lua:591)</td>
<td>-</td>
</tr>
<tr>
<td>97</td>
<td>0</td>
<td>loop</td>
<td>0.0%</td>
<td>100.0%</td>
<td>0.0%</td>
<td>(app)breathe:14 (core/app.lua:591)</td>
<td>-</td>
</tr>
<tr>
<td>143</td>
<td>0</td>
<td>-&gt;56</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>(app)breathe:14 (core/app.lua:591)</td>
<td>-</td>
</tr>
<tr>
<td>174</td>
<td>0</td>
<td>-&gt;116</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>(app)breathe:14 (core/app.lua:591)</td>
<td>-</td>
</tr>
<tr>
<td>87</td>
<td>0</td>
<td>loop</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>(app)breathe:14 (core/app.lua:591)</td>
<td>-</td>
</tr>
<tr>
<td>95</td>
<td>0</td>
<td>-&gt;63</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>(app)breathe:14 (core/app.lua:591)</td>
<td>-</td>
</tr>
</tbody>
</table>
New Features

Lot's of experimentation (open to evolving the language)

Add `jit.tracebarrier()` primitive #116

Merged by lukego merged 2 commits into raptorjit:master from lukego:jit-tracebarrier on Nov 7, 2017

Add `jit.unlikely()` primitive #143

Closed by lukego wants to merge 1 commit into raptorjit:master from lukego:trace-unlikely

Add `jit.seal(tab)` primitive (early version) #151

Closed by lukego wants to merge 1 commit into raptorjit:master from lukego:sealed
New Features

⚠️ Problem: Expensive heap-allocated boxes for 64-bit values (integers and pointers)

#91 opened on Aug 21, 2017 by lukego

🛠️ [WIP] Development branch for 96-bit VM

#199 opened on Nov 25, 2018 by lukego

64-bit values don’t fit into the VM’s 64-bit tagged words.

If they did, that would simplify a lot of things!
Future Goals

A weakness of LuaJIT are loops with unbiased branches.

In this paper the authors claim to solve that problem. We'd love to solve it for RaptorJIT!
Future Goals

Goal: Safe FFI memory access
#156 opened on Feb 1, 2018 by lukego

All (FFI) type information is available at runtime, and the JIT is really good at hoisting/eliminating guards/checks.

We want to try to provide memory safety for operations on C data by default!
Get Involved!

https://github.com/RaptorJIT/RaptorJIT

RAPTORJIT