LuaJIT on ARM64 - Status

Ryan S. Arnold, Zheng Xu

What is Lua?

- Lua is an interpreted scripting language that is easy to embed into application engines.
- Isn't Lua a language used for scripting video games?
- Originally yes, and...

Why Lua in the Enterprise?

- It's proven very effective as a scripting language for web server front-ends, for example NGINX: https://github.com/openresty/lua-nginx-module#readme
  - An embedded scripting language allows the execution of arbitrary and complex operations by the web server at runtime.
  - NGINX describes a number of typical uses for Lua: https://github.com/openresty/lua-nginx-module#typical-uses
    - For example, the NGINX web server use Lua coroutines to provide synchronous, yet non-blocking, API access to network services.
  - Lua provides compatibility with native C Languages programs through the Lua *Foreign Function Interface* (FFI).
- It's also showing up in software packet networking engines like Snabb.

Why LuaJIT specifically?

- LuaJIT is a Just-In-Time trace-compilation engine for the Lua language *(see Appendix A for a description of LuaJIT trace compilation).*

- Just-In-Time (JIT) trace-compilation is a performance optimization, whereby interpreted code that has been profiled and identified as a hot loop is compiled and then executed "just in time" on further iterations.

- LuaJIT (anecdotally) outperforms other (python, ruby, Lua interpreted) scripting languages by 10x.

- LuaJIT FFI (foreign-function interface) allows invocation of high-performance C-library functions from within Lua scripts.

Note: LuaJIT implements version 5.1 of the Lua language. Lua itself has moved on but LuaJIT will remain on 5.1.

LuaJIT JIT support on AArch64

- LuaJIT supports ARM64[1] in interpreted mode only, noticeably lacking JIT compilation mode.

- This puts enterprise AArch64 at a performance disadvantage to the x86_64 architecture.

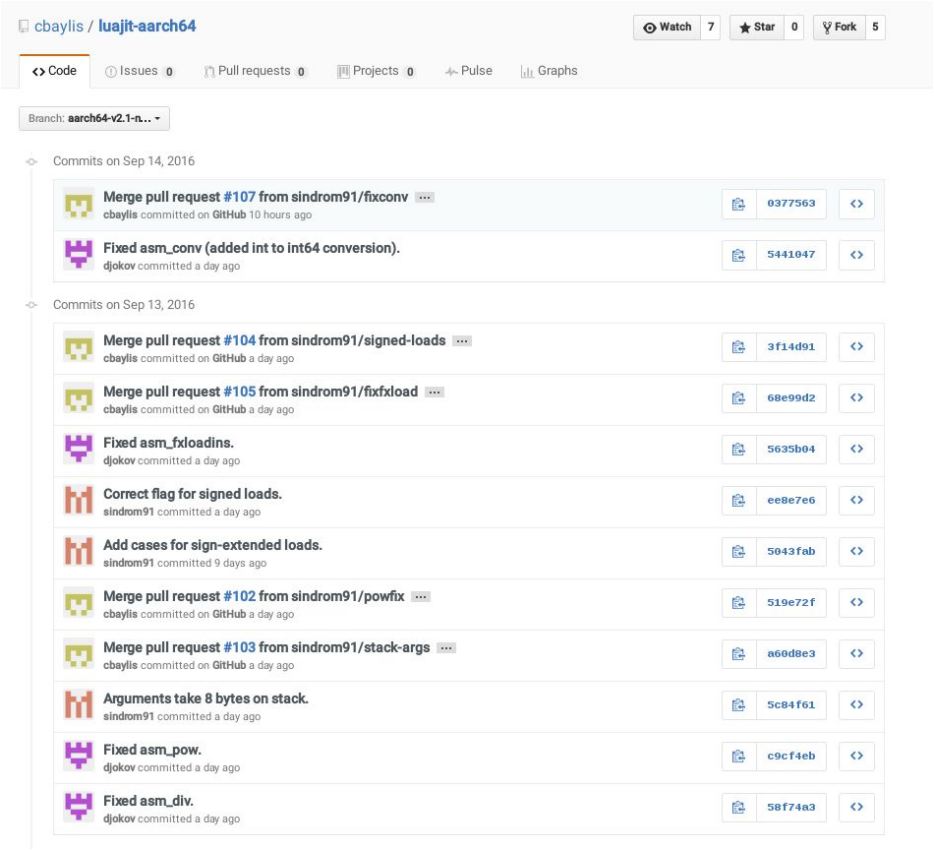
[1]LuaJIT refers to the 64-bit ARM-v8 architecture as *ARM64*

What's in a LuaJIT Architecture Port?

- Enable ARM64 JIT tracing & execution.
- Mark all ARM64 IR Translation Hooks as `lua_unimple()`
- Identify minimum viable program *(See Appendix B)*
- Implement Profiling and tracing hooks
- Implement necessary ARM64 IR Translation Hooks *(See Appendix A)* for MVP to execute to completion
- Foreign-Function Interface IR translation hooks
- Create tests to expose additional `lua_unimpl()` IR translation hooks.
- Testsuite Harness enablement
- Testsuite CI enablement
- Complete `lua_unimpl()`
- Integration Testing (NGINX)
- Verify LJ_GC64 implementation parity between ARM64 and x86_64.
- Optimization of ARM64 port
- ARM64 Disassembler

Linaro Involvement in LuaJIT

- On behalf of LEG, TCWG has been contributing to an upstream development effort with RT-RK, ARM, and Cavium, to enable ARM64 JIT support.
- Linaro merging pull requests: https://github.com/cbaylis/luajit-aarch64/commits/aarch64-v2.1-new
- Project development history at: https://github.com/cbaylis/luajit-aarch64/commits/aarch64-v2.1-new



Linaro connect
Las Vegas 2016

Phases of the LuaJIT ARM Porting Effort

The following phases of development were identified to signify major milestones and efforts in the porting process.

- **Phase 0** - Project Scope Definition [Completed May 2016]
- **Phase 1** - Minimum Viable Program Executing To Completion [Completed June 9, 2016]
- **Phase 2** - Expose and Implement remaining IR Translation Hooks [Completed September 12, 2016]
- **Phase 3** - Enable Continuous Integration & Integration Testing. Long tail of bug fixes and secondary feature fixes. [Projected for Year-End 2016]
- **Phase 4** - Upstreaming
- **Phase 5** - Optimization

ENGINEERS AND DEVICES WORKING TOGETHER

Linaro connect
Las Vegas 2016

Current Phase

The following phases of development were identified to signify major milestones and efforts in the porting process.

- **Phase 0** - Project Scope Definition **[Completed May 2016]**
- **Phase 1** - Minimum Viable Program Executing To Completion **[Completed June 9, 2016]**
- **Phase 2** - Expose and Implement remaining IR Translation Hooks **[Completed September 12, 2016?]**
- **Phase 3** - Enable Continuous Integration & Integration Testing. Long tail of bug fixes and secondary feature fixes. **[Projected for Year-End 2016]**
- **Phase 4** - Upstreaming
- **Phase 5** - Optimization

ENGINEERS
AND DEVICES
WORKING
TOGETHER

What Will Be Done In Phase 3?

- Phase 3 means that the long-tail of bug fixing has begun.
- We will implement automation (continuous integration) to prevent functional regressions when new fixes are introduced.
- We will use integration testing to verify that ever more functionality is available in real-world applications.
- We will verify that there is parity in the LJ_GC64 implementation between ARM64 and x86_64 to prove that 64-bit addressability is complete.

What Was Completed in Phase 2?

- **The majority of the porting effort is in exposing (with test cases) the unimplemented IR translation hooks and implementing them.**
- New micro-tests were created to expose as many IR translation hooks as possible.
- Eventually it was no longer easy to implement new micro-tests so fuller feature LuaJIT testsuite functions were extracted and run in isolation to expose further IR translation hooks.
- Solving some 'bugs' had a cascade effect where a number of failures now succeed, e.g., ARM64 asm_href was responsible for 24 testcase failures.
- The testsuite framework can now execute correctly (but there is a segfault during garbage collection that crashes the testsuite unpredictably).

What's Left For The ARM64 LuaJIT Architecture Port?

- Enable ARM64 JIT tracing & execution.
- Mark all ARM64 IR Translation Hooks as `lua_unimple()`
- Identify minimum viable program *(See Appendix B)*
- Implement Profiling and tracing hooks
- Implement necessary ARM64 IR Translation Hooks *(See Appendix A)* for MVP to execute to completion
- Foreign-Function Interface IR translation hooks
- Create tests to expose additional `lua_unimpl()` IR translation hooks.
- Testsuite Harness enablement
- Testsuite CI enablement
- Complete `lua_unimpl()`
- Integration Testing (NGINX)
- Verify LJ_GC64 implementation parity between ARM64 and x86_64.
- Optimization of ARM64 port
- ARM64 Disassembler

Current ARM64 Port Functional Completeness

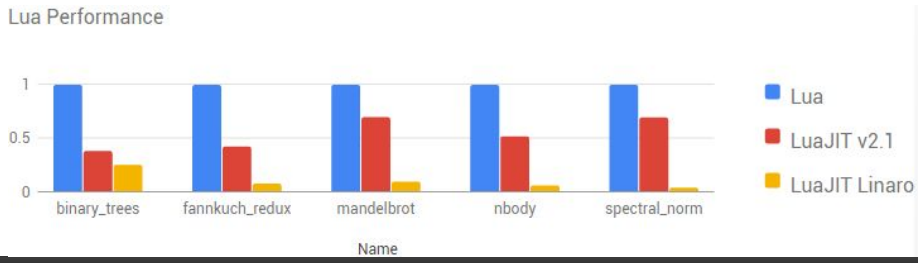
- The optimization IR translation hook is the only remaining `lua_unimpl()` in the code-base from a starting list of 64.
- All testsuite tests can be executed in isolation.
- Garbage Collector bug presently impeding execution of the testsuite. Testsuite crashes with GC segfault after 212 tests.
- FFI translation hooks are complete
- Test-suite parity between ARM64 JIT and Interpreted mode
- 3 failures more than on x86_64. *(See Appendix C & D)*
- Github CI commit triggered testsuite execution not yet started.
- Linaro CI to integration test NGINX with LuaJIT changes in progress.
- LJ_GC64 implementation parity with x86_64.

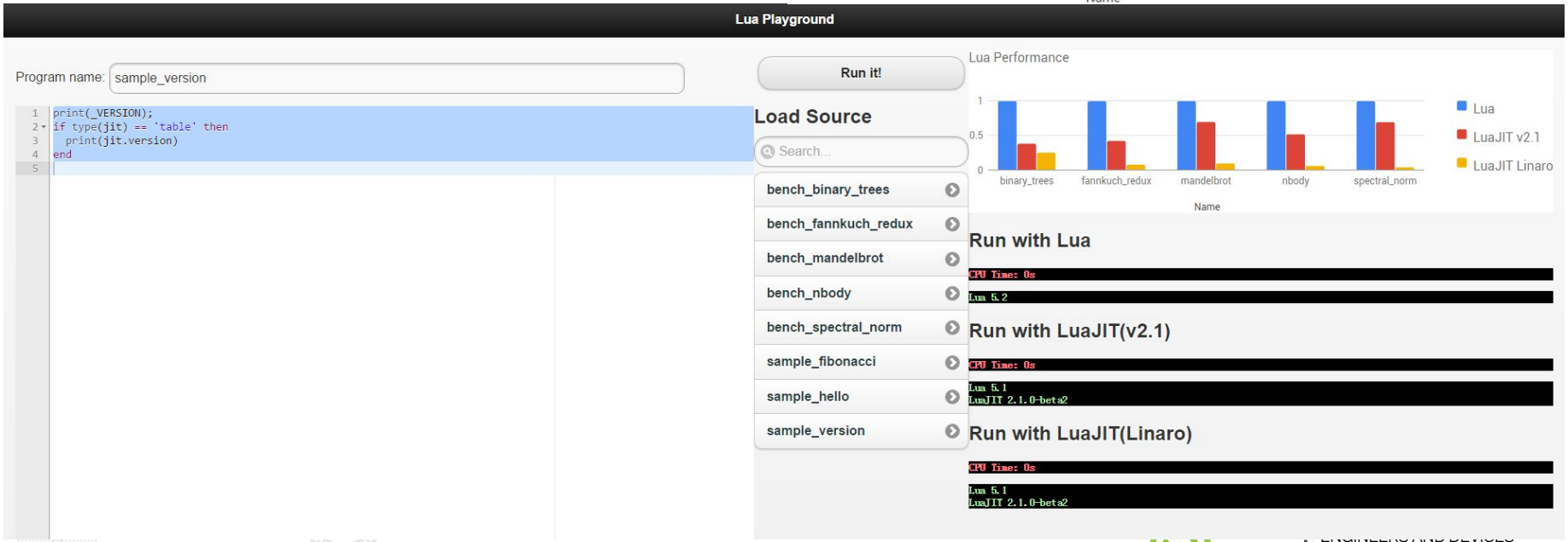
Schedule

- Original estimation was that IR translation hooks might be completed by end-of-september. The team has implemented all but the optimization hook which will be implemented in phase 5.
- Testsuite CI enablement should be enabled by Mid Sept.
- Integration CI should be enabled by the end of Oct.
- We need to verify LJ_GC64 implementation parity with x86_64 (Mid November?)

LuaJIT on ARM64 demo

- http://64.28.99.85/
- Run Lua program on aarch64 with different implementation.





Useful links on ARM64 port

- GitHub project : https://github.com/cbaylis/luajit-aarch64
- Linaro CI : https://ci.linaro.org/view/luajit/
- Mailing list : https://lists.linaro.org/mailman/listinfo/luajit
- Lua playground : http://64.28.99.85/

Appendix A: How does JIT Trace-Compilation work?

- Most Lua code is executed by the LuaJIT interpreter.
- During execution the engine profiles, and performs a trace of the program and records execution information in LuaJIT bytecode.
- LuaJIT Intermediate Representation (IR) is emitted from that bytecode.
  - LuaJIT IR consists of types and instruction definitions such as:

    arithmetic/conversion/comparison routines, constants, bitops, overflow ops, memory ops, loads and stores, barriers, and function call forms, et al.
- A *hot loop*[1] is identified by profiling
- The compilation phase of JIT execution calls architecture specific *translation hooks* to generate machine code from the recorded IR for the hot loops.
- Further iterations of the hot loop execute the compiled code.

[1] A *hot loop* is a loop where profiling has determined the program spends most of its execution time.

Appendix B: Minimum Viable Program?

- The following is a minimum viable program that profiles as a hot loop:

```
print("Hello World")
t = 0
for i = 1,100 do
    t = t + i
end
print(t)
```

Appendix C: x86_64 Testsuite Results

```
x86_64 :
[20:59:10]ent-x86-01-ubuntu-luajit-test-cleanup:MSG: ########################
[20:59:10]ent-x86-01-ubuntu-luajit-test-cleanup:MSG: ## [Failed Tests : 5] ##
[20:59:10]ent-x86-01-ubuntu-luajit-test-cleanup:MSG: ########################
[20:59:10]ent-x86-01-ubuntu-luajit-test-cleanup:DBG: Running : cat
/home/ent-user/ci-scripts/fail.log
lib/table/pack.lua
lib/base/pairs.lua
lib/contents.lua
misc/stack_purge.lua
lang/goto.lua
```

Appendix D: ARM64 Testsuite Results

```
arm64 :
[21:01:29]ent-arm-02-ubuntu-luajit-test-cleanup:MSG: #######################
[21:01:29]ent-arm-02-ubuntu-luajit-test-cleanup:MSG: ## [Failed Tests : 8] ##
[21:01:29]ent-arm-02-ubuntu-luajit-test-cleanup:MSG: #######################
[21:01:29]ent-arm-02-ubuntu-luajit-test-cleanup:DBG: Running : cat
/home/ent-user/ci-scripts/fail.log
lang/goto.lua
lib/base/pairs.lua
lib/contents.lua
lib/table/pack.lua
misc/catch_wrap.lua
misc/stack_purge.lua
sysdep/catch_cpp.lua
unportable/ffi_arith_int64.lua
```

Linaro connect
Las Vegas 2016

Thank You

For questions on this presentation contact:
Ryan S. Arnold <ryan.arnold@linaro.org>

#LAS16
For further information: www.linaro.org
LAS16 keynotes and videos on: connect.linaro.org