

Scripting at the Speed of Thought



Using Lua in C++ with sol3

September 28th, 2018 - CppCon, Meydenbauer Center



[@thephantomderp](https://twitter.com/thephantomderp) [#sol3an](https://twitter.com/sol3an)

Scripting

- Treating code as data
 - Means of extending your application beyond ship time
- Delegate smaller, simpler tasks to easy-to-use language

Why Script?

- Extending your application
 - without re-compilation
 - without fragile DLLs/shared libraries
- Lowering barrier to entry
 - Complexity of main language (C++, Java, Haskell) versus complexity of scripting language (Lua, Python, Javascript)
 - Little / no IDE or build setup required

“What about the Standard?”



- There is no scripting or interop built into the standard library
 - ... at all
- Will compare to other libraries, however!

Lua, the Scripting Language



- Simple dynamic language
 - Grammar fits on a single page
 - Only 1 data structure - table
 - Minimalistic library
- Famously used in
 - World of Warcraft (WoW), Redis database, Operating System components, Servers, Waze GUI, ...

Lua, with C code

- Entire VM is tiny and compiled with ANSI C
 - Portable to many, many compilers
- Exposes library interface, the Lua C API
 - Stack-based: push values, call manipulation operations, pop leftover values
 - Fairly fast compared to other VMs

Lua C API: how-to-stack

- `my_table["a"]`
 - get `my_table` global
 - get field
 - `lua_to{x}` value
- `my_func(2)`
 - push `my_func` global function
 - push argument
 - call, get return(s) using `lua_to{x}`

Okay, let's scale it up...

```
other_func( my_table["a"]["b"], my_func( 2 ) )
```




Miss me with that noise, fam.



Limitations of C itself

- No overloading - “which one do I need, again?”

```
lua_gettable()  
lua_getglobal(const char*)  
lua_getfield(const char*)
```

```
lua_geti(int) // version 5.3+  
lua_rawgeti(int)  
lua_rawget()  
lua_rawgetp(void*)
```

Wrap/Abstract the C Code

Stack Namespace

- `sol::stack` namespace that gets rid of all the C code
 - `stack::push(v)` -takes any value and pushes it with appropriate API
 - `stack::get<T>` -takes any type and retrieves it with appropriate API
 - `stack::check<T>` -takes any type and reports if it exists
- “Type Rich” Programming
 - Tells us how to push, how to get based on type / argument type
 - Improves safety, increases developer throughput

Stack Namespace - Composed Operations

- `stack::get_field(key, table_index)`
 - Optimize to `lua_getfield` if key is a string (or string-like)
 - Optimize to `lua_getglobal` if we are working with global table
 - Otherwise fallback to default `lua_gettable`
- `stack::set_field(key, value, table_index)`
 - Optimize to `lua_setfield` if key is a string (or string-like)
 - Exactly like above

An example

```
lua_State* L = ...;
sol::stack::get_field<true>(L, "some_key");
int the_value = sol::stack::get<int>(L, -1);
lua_pop(L, 1);

lua_createtable(L, 0, 2);
sol::stack_reference ref(L, -1);
sol::stack::set_field(L, 1, "val1");
sol::stack::set_field(L, 2, "val2", ref.stack_index());
ref.pop();
```

Not High Enough

- Still working with the stack, stack indices
- This is good for a C developer, but what about a newcomer?
- C++, have reusable data structures - mimic language:
 - table
 - function
 - userdata

Improving the Abstraction



reference and the rule of zero

Abstraction Layer 0

- Lua has a “C registry”
 - Place where C API user can store references to Lua constructs
 - References are counted: only freed when reference count hits 0
- Want table, function, thread, etc.
 - All must be reference counted in C and C++ code!

Abstraction Layer 0: Rule of 0

- `sol::reference`
 - reference counting abstraction
 - Slightly similar to `boost::intrusive_ptr`
 - More accurate to proposed `std::retain_ptr` or ARC references
 - wg21.link/p0468
- Increments on copy
- Reference-steals on move
- Decrements on destruct

Abstraction Layer 1: Reuse

- class `object` : `sol::reference` { ... };
 - Just add `.as<T>()` and `.is<T>()`
- class `function` : `sol::reference` { ... };
 - Just add `.call()` and `operator()(...)`
- class `table` : `sol::reference` { ... };
 - Just add `.set()` and `.get()` and `operator[](...)`

Abstraction Layer 2: Proxy

- Need `int x = f(obj)` and `my_class& y = f(1, 2)`
 - Do not want to make 2 different types
 - Simple struct that has a templated implicit conversion operator
 - struct `proxy { template <typename T> operator T () { ... } }`;
- `my_table["foo"] = "bar"; std::string bar = my_table["foo"];`
 - Same as above, plus:
 - `operator=` to handle assignment from anything
 - `operator[]` to “chain” lookup with `int x = my_table["bark"]["bjork"];`

Abstraction Layer 2: Problems

- “Unicorn” Proxies are weak to certain idioms:
- `int a, b, c;`
`std::tie(a, b, c) = f();`
 - Asks for `std::tuple<int&, int&, int&>`
- Cannot control return type of implicit or explicit conversions!
 - No way to hand back `std::tuple<int, int, int>`

Abstraction Layer 2: Fixes?

- Relevant fix paper for C++ San Diego meeting
 - p1193, Explicit Return Types for Implicit Conversion
- struct proxy {
 - template <typename T>
 - handle_weird_tie_stuff_t<T> operator T () { ... }
- };

Usertypes

The Glory of Sol

Usertypes are A M A Z I N G

- The primary binding glue between C++ and Lua
- Exposes C++ classes, their idioms, their properties cleanly and efficiently
- But why take my word for it? Let's just have a quick peek...

DEMO IN PROGRESS



But There's More!

- Usertypes - and sol2 in general - can also work with *types and systems it does not know about*
- Put sol in your large commercial codebase, today
 - Use it incrementally without problem thanks to `sol::state_view`
 - Transition as fast or as slow as you desire



Xottab_DUTY 09/09/2018

I also need to say thanks for `sol::state_view`. Currently, I'm just using both `luabind` and `sol2` and it works perfectly!

DEMO IN PROGRESS



It is FEATURE PACKED!

- Some people don't even realize...

Falco Girgis 09/17/2018 11:42 AM

Ugh.
I'm about to have to hack some crap into OOLua...
I just don't have time to switch just yet...

@ThePhD We rely heavily on special like SSE/Neon/SIMD accelerated Vector/Matrix type proxies to Lua for a bunch of math in our scripts... these things are sooo much prettier and easier with overloaded operators... but do you allow of overloading the overloaded operators? Like Vector2 for example is divisible by both another Vector2 and a scalar... I have a scenario like this (trivial case, there can be a looot of linear algebra going on in these scripts):

```
--Clear out the range
local start = (Vector2:new(origin) - Vector2:new(radius))/32
local stop = (Vector2:new(origin) + Vector2:new(radius))/32
```

Of course since in OOLua I implemented the arithmetic operators as Vector2 vs Vector2 types, yeah, it's not implemented for ints... so dividing by 32 there is expecting a Vector2 obvious solution is this:

```
--Clear out the range
local start = (Vector2:new(origin) -
Vector2:new(radius))/Vector2:new(32)
local stop = (Vector2:new(origin) +
Vector2:new(radius))/Vector2:new(32)
```

but honestly that's really really really bad for us in terms of performance. All of these tiny little in-line heap allocations Lua loves to do fragments the shit out of our 16MB of RAM on the Dreamcast, so we really try not to do that.
So basically what I do in OOLua for stuff like this...

Is I've hacked in my "LuaVariant" type as like a first-class citizen in OOLua... It treats it like a Lua primitive type and just knows how to push and pop... Then I implement things like this:

```
inline Vector2 Vector2::operator/(LuaVariant variant) const {
    switch(variant:getType()) {
        case LUA_TNUMBER: return *this/Vector2(variant:getValue<float>());
        case LUA_TUSERDATA:
            if(variant:getTypeName() == "Vector2") return
            *this/variant:getValue<Vector2>();
            default: break; //you dun fucked up
    }
    return Vector2(0,0f);
}
```

yeaaaaah, it's crazy, but my variant system really lets me do whatever-the-hell I want so I can dance around OOLua's shortcomings, because it's really powerful.
But it's also inconvenient to have to do this type of things manually. lol.
That's seriously how I basically support function overloading in Lua by type, C++-style.
Basically I OOLua is just for registering types and methods easily and then I literally just

Amos 09/17/2018

<https://la.wentropy.com/3aGX>

this is extremely slow.... 1000x slower than native implementation
um, how can I optimize it
the code is simply an bfs calculation

```
function bfs(s)
    local frontier = create_vector(1)
    frontier[1] = s
    local visited = create_flags(vertex_num)
    local cnt = 0
    while 0 < #frontier do
        cnt = cnt + 1
        if (cnt > 1000)
            return
        end
        next = create_vector(0)
        for idx = 1, #frontier do
            local node = frontier[idx]
            visited[node] = true
            local o = out(node)
            for i = 1, #o do
                local v = o[i]
                if visited[v] == 0
                    then
                        visited[v] = 1
                        next:add(v)
                    end
                end
            end
            frontier = next
        end
    end
end
```

I'm using a bunch of vector in lua. I observed a lot of overheads when random accessing
is it possible to get native dereference speed on lua side?

moka 09/19/2018

is there any recommended way to expose custom iterator ranges to lua with sol? I.e. I have some form of a container view that adds additional meta information to the underlying data that it is iterating over. The iterators in question only dereference to value types and hence won't compile with the existing container (as that expects const T & or T & for * operator).
My current work around is to simply create a table in those cases which is ok but not that nice for a lot of data.

Witnessing Realizations in Real-time is fun



Falco Girgis Last Tuesday at 4:58 AM

DUUUDE

he supports overloading overloaded operators out-of-the-box!?

Wow, thanks.

I thought this was such an exotic request... and it's already there. Imao

I appreciate it!



moka 09/20/2018

awesome, thank you!!



moka 09/20/2018

got it to work, appreciate all the useful pointers and examples!



Kisu 09/14/2018

@ThePhD thanks for the tip about using the `initializer_list` version! it brought my executable size almost back to what it was and now it's not taking 5 minutes to link lol

Customization Points

And the problem with Defaults

Past Customization: Struct Specializations

- Specialize a template struct getter/checker/pusher
 - Works and scales
 - Users can add their own specializations

The structs below are already overridden for a handful of types. If you try to mess with them, you'll get thick template error traces and headaches. Overriding them for your own user de

struct: getter

```
template <typename T, typename = void>
struct getter {
    static T get (lua_State* L, int index, records tracking) {
        // ...
        return // T, or something related to T.
    }
};
```

This is an SFINAE-friendly struct that is meant to expose static function `get` that re

Past Customizations: Problem with Structs

- Users do not like way e.g. `int64_t` or `uint64_t` or a container are handled
 - Impossible for them to extend if they are not careful of mutually exclusive SFINAE

```
template <typename T> <T>
struct pusher<T*, std::enable_if_t<
    meta::all<is_container<meta::unqualified_t<T>>,
    meta::neg<is_lua_reference<meta::unqualified_t<T>>>>::value
>> {
    typedef std::add_pointer_t<meta::unqualified_t<std::remove_pointer_t<T>>>> C;

    static int push(lua_State* L, T* cont) {
        stack_detail::metatable_setup<C> fx(L);
        return pusher<detail::as_pointer_tag<T>>{}.push_fx(L, fx, cont);
    }
};
```


Customization Layer Mk. II

- Customization points are instead functions
 - attempt to call via ADL and priority tags (Thanks, Arthur O'Dwyer!)
- `sol_unqualified_get(priority_tag, types<custom_string>, ...);`
 - Hand user a priority tag type with higher ranking:
 - using `priority_tag = max_priority;`
- Keep struct specializations for “fallback” and “defaults”
 - Clean separation

Customization Layer Mk. II: Compile Harder

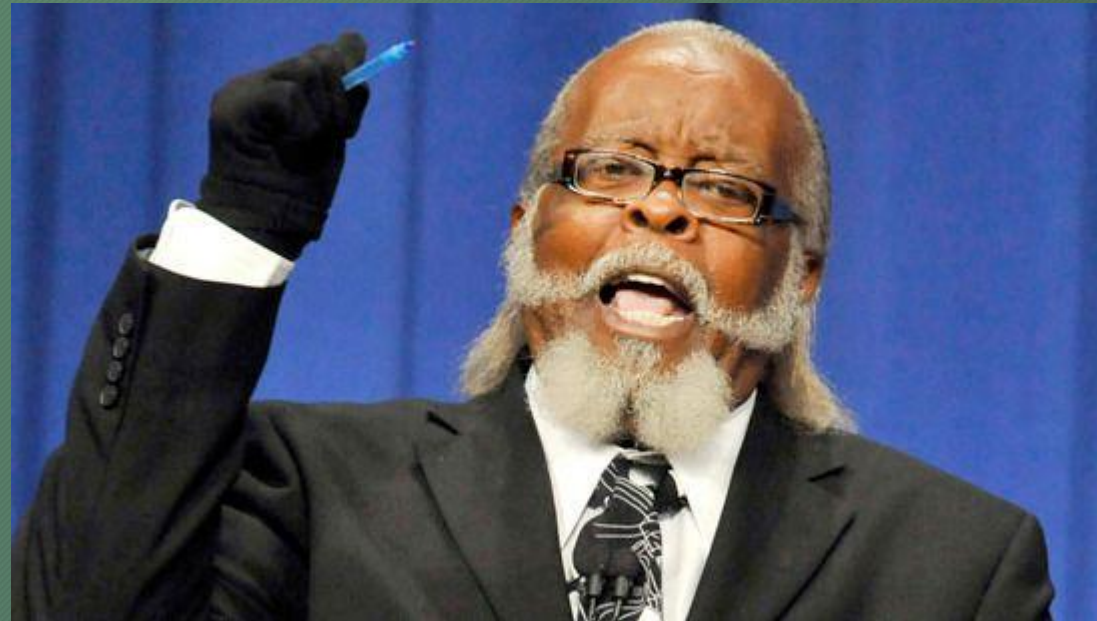


- N.B.: they are just regular functions
 - Implement in cpp files, export them
 - Compile heckin' fast
 - Everyone knows how to write a function (!!)
- Users do not have to understand SFINAE or template struct specialization
 - Vastly smaller amount of people SFINAE and struct specialize
 - Decreases barrier to entry

Speaking of Compile Times...

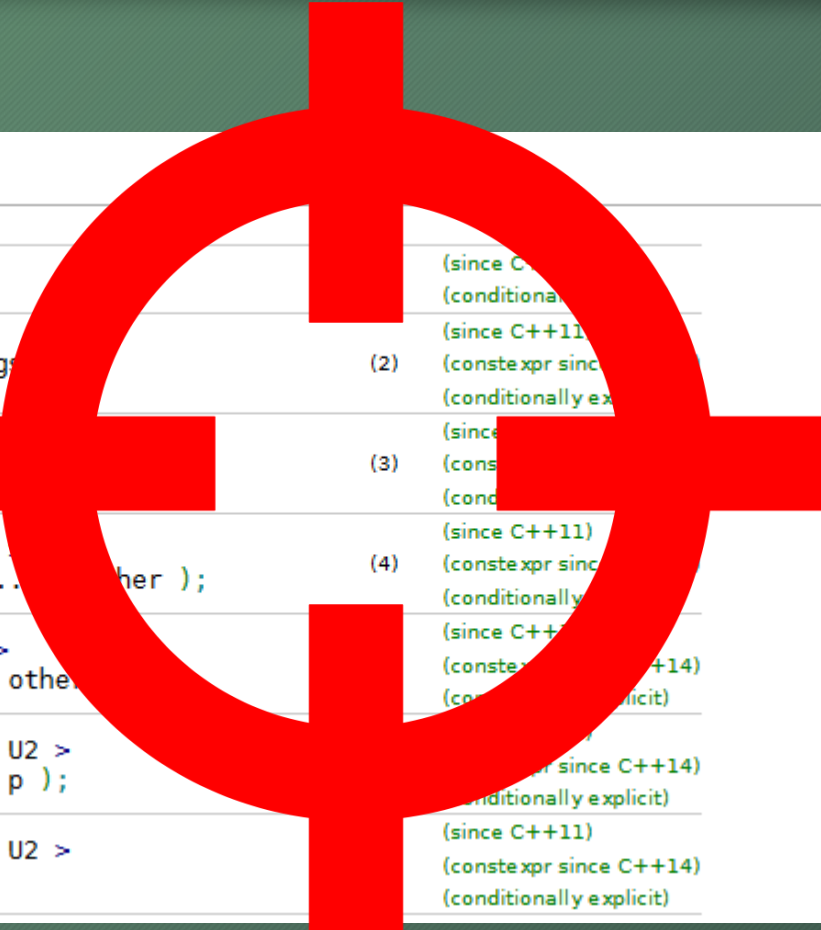
Compile Times + Compiler Memory Usage

- Too. Damn. High.



Tuple and SFINAE: the Greater Evils

```
std::tuple::tuple
Defined in header <tuple>
constexpr tuple();
tuple( const Types&... args );
template< class... UTypes...>
tuple( UTypes&&... );
template< class... UTypes...>
tuple( const tuple<UTypes...> &other );
template <class... UTypes>
tuple( tuple<UTypes...>&& other );
template< class U1, class U2 >
tuple( const pair<U1,U2>& p );
template< class U1, class U2 >
tuple( pair<U1,U2>&& p );
```



A Spark of Moonlit Inspiration...



- “Can I keep the same speed and still remove tuple and a lot of compile time information”?
 - Benchmark cpp file for sol3 took about 15% less time than sol2 benchmark file
 - But did we keep the same performance?!

So I implemented it. Today.



- In the dark of the night. Just before my presentation.
- The “I Like Anxiety” Challenge:
 - Ran benchmarks overnight and promised myself that I would modify presentation with the numbers in the morning

Idea: Virtual Base class, templated data?



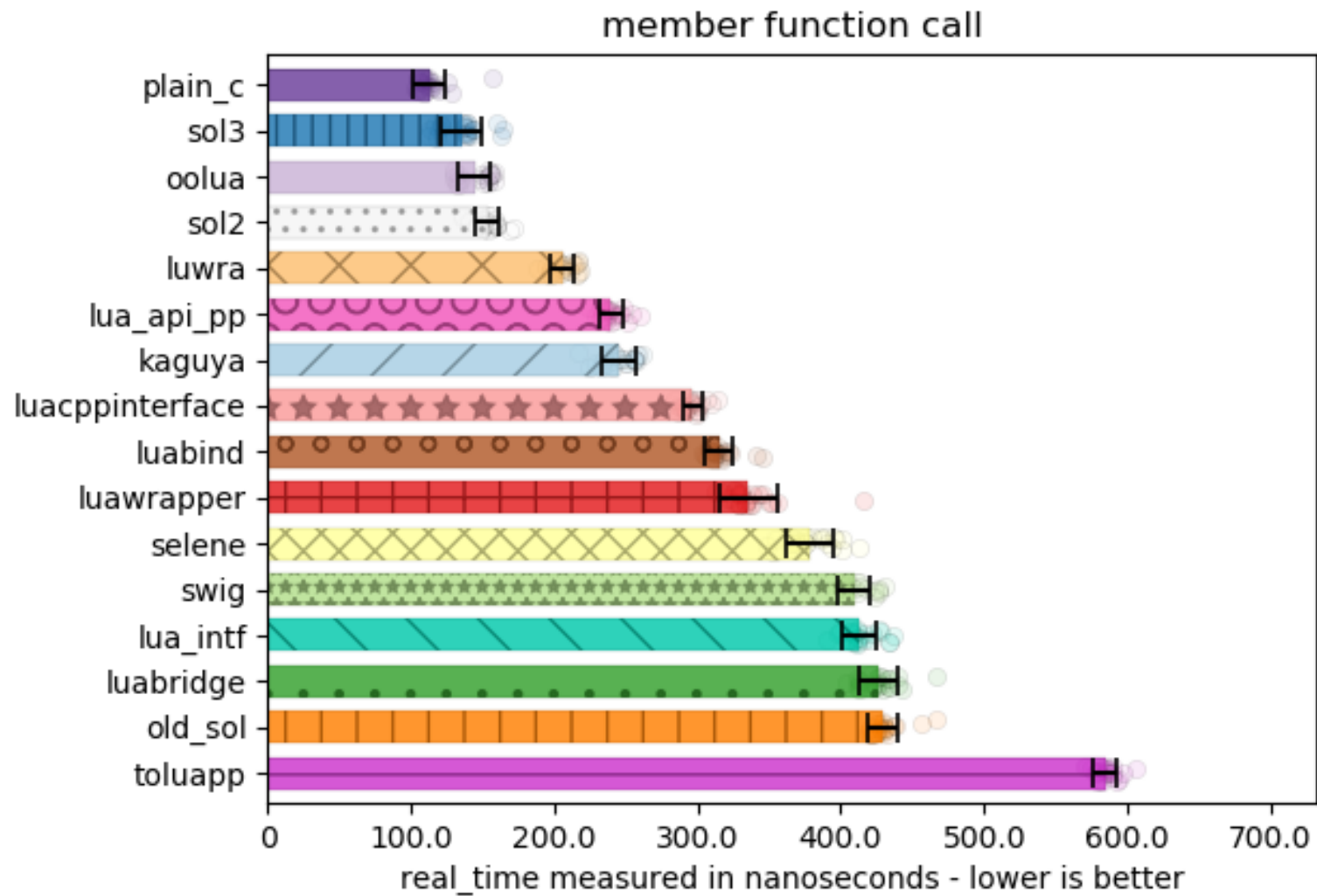
- Essentially:
 - `std::vector<std::unique_ptr<binding_base>>`
- Do 4 things:
 - Store data in vector to give it a never-changing memory address
 - Get data as `void*` to transport through Lua + templated free function
 - Make sure either side of Lua abstraction preserves type information
 - Pass along the *exact address*



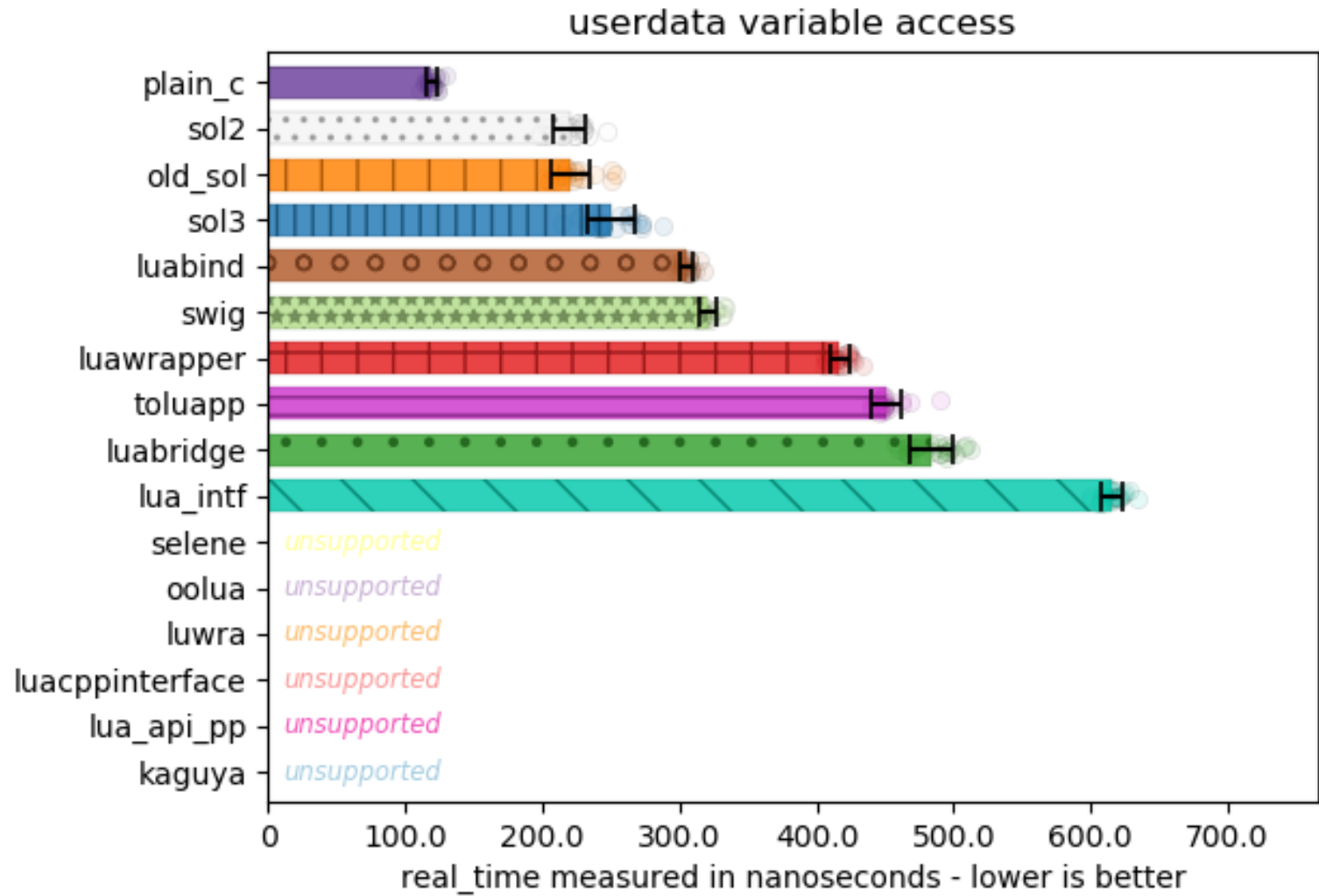
Sun Came Up, Benchmarks Finished...



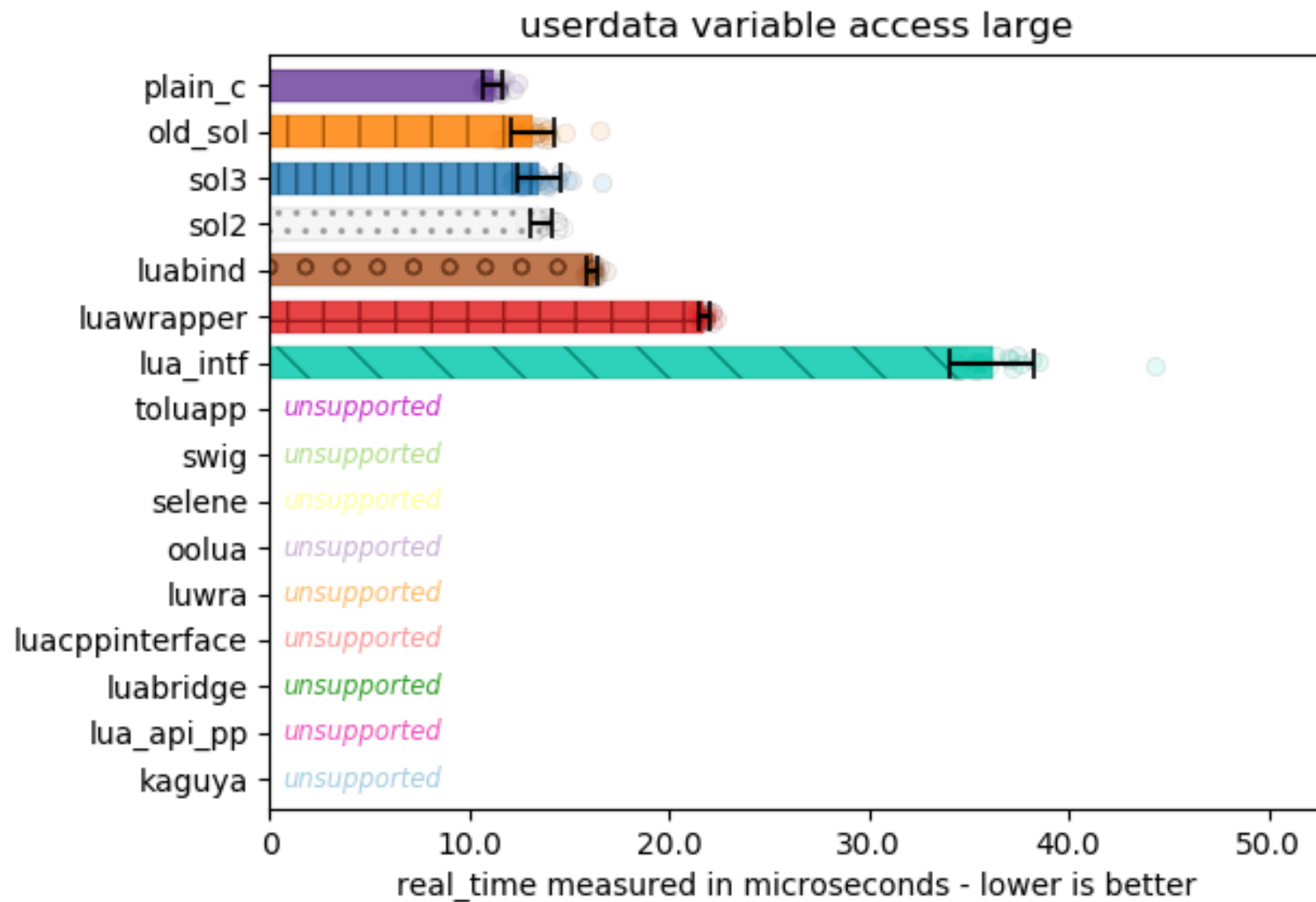
Nice.



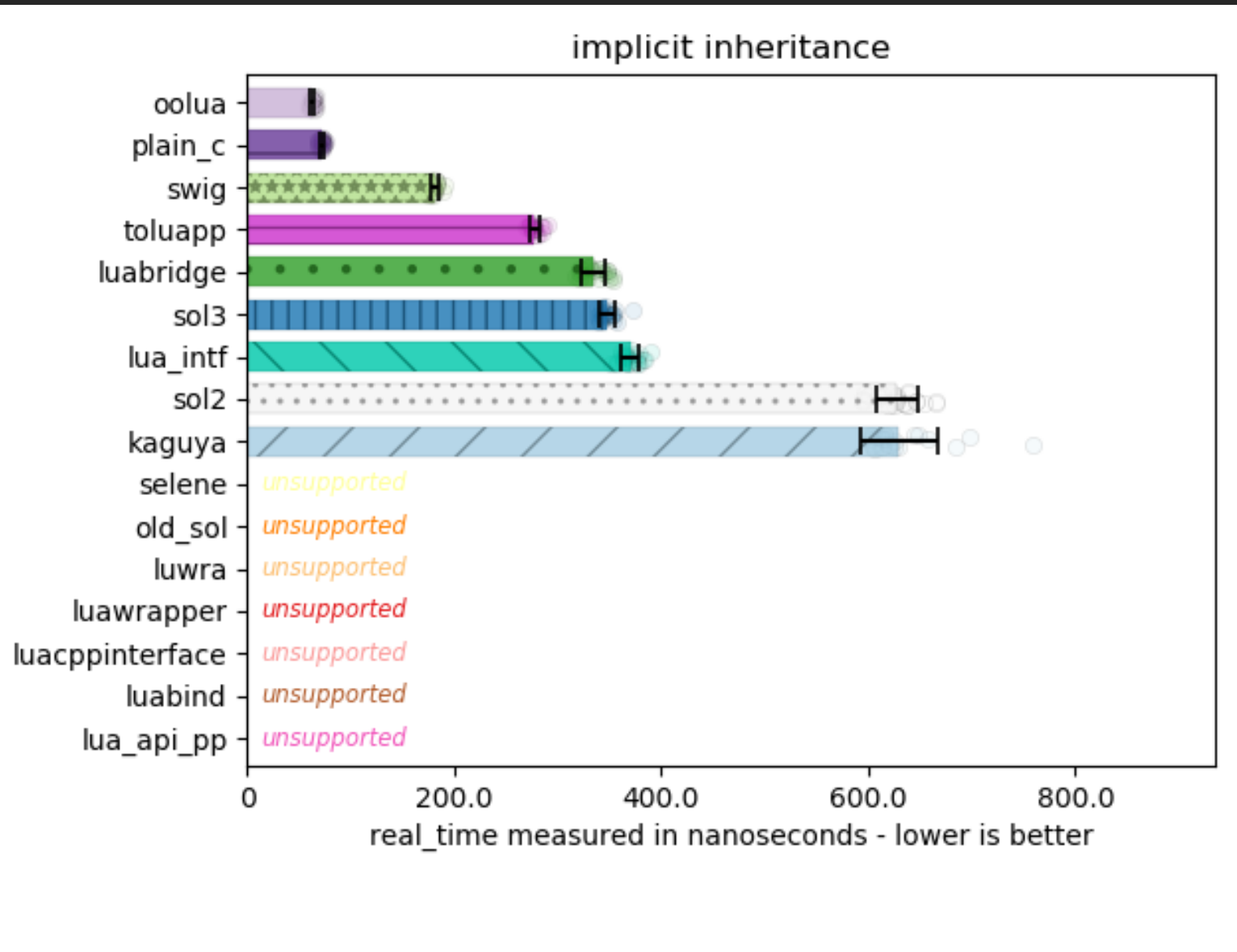
Nice.



Nice.



Nice.



Nailed It

- Great performance metrics
 - So much room to decrease compile times
 - So many more features enabled by this exploration and the advancements!
- sol3 pushed to its own branch
 - Until tests are converted and all pass
 - Hope to have something by the end of October, before the C++ San Diego meeting starts

My felt gratitude...

- #include - for the graphs and being a wonderful community
 - Inspired a Python Talk
- Companies and individuals who have used sol2 to success and have recommended it to others!
 - Corentin, Elias Daler, Orfeasz, Xottab_DUTY, and donators up to this point!
 - Jason Turner for telling me to start talking about sol2 and sol3!
- Mother Eugenie, Sister Lorigiana



[@thephantomderp](https://twitter.com/thephantomderp) #sol3an



<https://www.patreon.com/thephd>



<https://www.linkedin.com/in/thephd>



<https://github.com/ThePhD/>
<https://github.com/ThePhD/sol2>

Thank YOU for coming!

It's hard to choose against Sean Parent and Hyrum Wright and Matt Godbolt and Odin Holmes! So thank you for making my first CppCon talk a great one!

