An Analysis of Various LUA Class Implementations

Evan Liang
Stanford University

Abstract

Classes are an integral part of C and C++, yet completely lacking from the basic Lua library. There are many different tradeoffs and improvements that must be considered when building a class implementation, such as memory usage vs speed, optimizing instance creation vs data access, and whether or not building an accelerated C version is worth it. As it turns out, forgoing memory constraints and allowing each class to retain its own copy of the virtual method table can result in runtime improvements of up to 9-10 times, while utilizing a custom external C implementation can result in runtime improvements of a factor of 2-3 times. Which improvements are worth it will change for every use case, but the quantitative data comparing them is all here to help make the right decision.

I. Background

Classes are a familiar concept to those that have worked with Object-Oriented Programming languages before. A class is essentially a template for creating objects, allowing the programmer to define a single set of methods and state and then reuse it. However, as a relatively light scripting language, Lua does not come with its own built in class implementation. The goal of this project is to build several class implementations in Lua and analyze the tradeoffs made between expressiveness and performance in each.

II. Approach

First, we define the class system that we are going to build. The basic class system is a single-inheritance model with public methods and public data members. Private data members were excluded due to the difficulty of implementing them without introducing security vulnerabilities (i.e. ensuring that they absolutely cannot be modified outside of the class). Metamethods are classified under methods and therefore are also inherited from parent to child.

i. Design One

This design is virtually equivalent to the assignment two design. Children maintain references to the their parent and metatables are used to define fallbacks. That is, if a method or data member is not found in a child, then the lookup continues within the parent, and so on, until the base Object class is reached. The only exception to this are the metamethods, for which each class stores all of its own metamethods in addition to all the metamethods from all its parents, grandparents, and so on.... Therefore, this design stores minimal information, as only the data members and methods unique to the child are stored within the child object (with the exception of metamethods).

ii. Design Two

In C++, when a class defines a virtual function that can be overwritten in a child class, most compilers will use something called a vtable for lookups. This vtable (short for virtual method table) is an array of pointers to the virtual functions. When a subclass inherits from a superclass, it simply copies the superclass's vtable, rewriting the pointers to virtual function that it overwrites, and adding any pointers to virtual functions that it itself may define. This design utilizes much more space per class than Design One, since each class now has its own table of pointers to the inherited functions, instead of only its own functions, and then performing function calls to inherited functions by traversing up the inheritance tree.

Because Lua stores references to functions inside of its tables, an approximation of this design can be created in Lua by making a copy of the parent's methods and then adding in the child's methods (to modify function references to overwritten methods and add methods specific to the child). This is essentially the same way metamethods were inherited in Design One, but now extended to methods as well. Method calls of inherited methods are much faster in this implementation since it no longer has to fall back to the parents methods (and potentially grandparent, and so on).

iii. Design Three

Most Lua class implementations will make use of Lua metatables to define fallbacks for method calls. However, one other thing to consider is how to deal with inherited data members. In the naive Design One implementation, data members were treated similar to methods. If access were requested for a data member, and it was not found in the child class, then lookup of the data member continued in the parent class.

This design, based off of the "Yet Another Class Implementation" (cited in References), takes advantage of the fact that we don't actually have to perform this lookup! Since there is no concept of private variables and therefore no need to check whether or not modifying a particular data member is valid, there is no need to perform this lookup any more. Instead, any data value that requires initialization can be initialized in the constructor (which also means that the parents constructors and so on... may need to be called as well). Any data member that does not require initialization can just be created when first referenced.

iv. Design Four

As a scripting language, Lua, by itself, does not provide the programmer the control necessary to perform micro optimizations compared to a language like C or C++. Luckily, Lua's C API allows us to create a high performant C implementation of the class system that can then be used within Lua.

The design of this implementation is virtually identical to Design One, except now implemented in C. Each class is represented as a userdata object, containing a name to uniquely identify the class, and then a methods and data table.

The function to create a new class takes in three parameters, a unique name for the class, the unique name associated with its parent class ("object" if none), and then a table of methods (which may include a constructor). There is no parameter for a table of data members since the data members can be created lazily (i.e. in the constructor or when first referenced).

With these three parameters, we now have enough information to setup the new class. We begin by adding a "new" function to the passed in methods class, which the user can call to create new instances of the class. We then grab the metatable associated with the parent class (using the luaL_getmetatable function), and then assign it as the metatable for the passed in methods table. We finish up by creating a new metatable for the child class, and setting the "__index" value to the passed in methods table. This is sufficient to define all the fallback needed to method calls.

This design was by far the most difficult to implement due to having to deal with the virtual stack and having to chain multiple API calls to achieve a simple request. Of course, wading through all those unhelpful segmentation fault messages are worth it in the end, because this kind of control is precisely what allows us to write a more efficient class implementation that is devoid of the excess commands that are not necessary.

v. Testing Framework

We created a testing framework in order to evaluate the performance of each class implementation. The following aspects of each class were tested: instance creation, method call (for inherited and non-inherited methods), data access (for inherited and non-inherited data), and metamethod access (for inherited and non-inherited metamethods). Over 50 000 individual trials were run for each measurement, and a cumulative time for all trials was recorded. There were no tests for the creation of the class table since there is no realistic situation in which we would define so many new classes as to make this runtime significant.

III. Results

We use Design One, the original assignment two class implementation, as the basis of comparison for the rest of the designs. The following figures depict the performance of Design One.

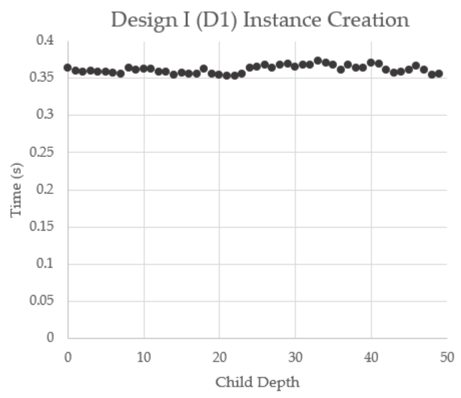

Figure 1: *The effects of child depth on instance creation in Design 1*

Child depth can be defined inductively in the following manner:

1. The base class which inherits only from the object class has child depth 0.
2. For any pair of child and parent classes, the child has a child depth of 1 greater than the child depth of the parent.

From Figure 1, we can see that for Design 1, the number of parents, grandparents, etc. that a class has does not impact the time it takes to create a new instance of that class.

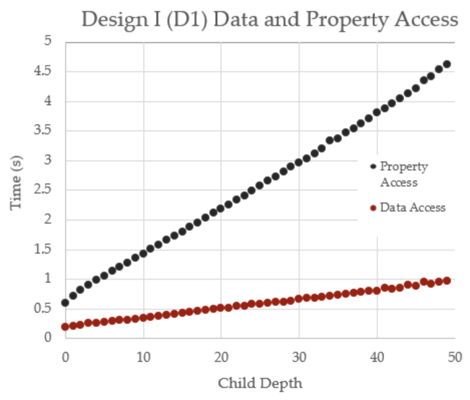

Figure 2: *The effects of child depth on the method calls and data access*

Figure 2 depicts a roughly linear relationship between child depth and the time required for method calls and data access. This makes sense since Design 1 utilizes fallbacks to implement inheritance, and therefore for each additional ancestor a class has, there is one additional set of methods and data members to search through.

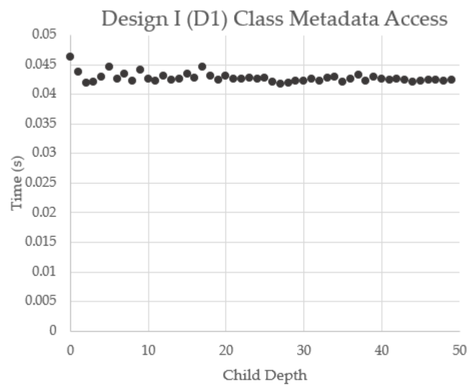

Figure 3: *The effects of child depth on the metamethod calls*

Since Lua does not allow metatables to have metatables, all of our implementations requires each class to have its own copy of all of its metamethods as well as all of its parents metamethods. This means that child depth does not have an affect on the runtime of calling metamethods. This implementation strategy is consistent across all of our class designs, and therefore, the graphs for metamethod calls using Design 2 (D2), Design 3 (D3), and Design 4 (D4) all look virtually identical to this one.

i. Design Two Results

As mentioned in our approach section, we expect the main difference between Design Two and Design One to be a reduced call time for inherited methods.

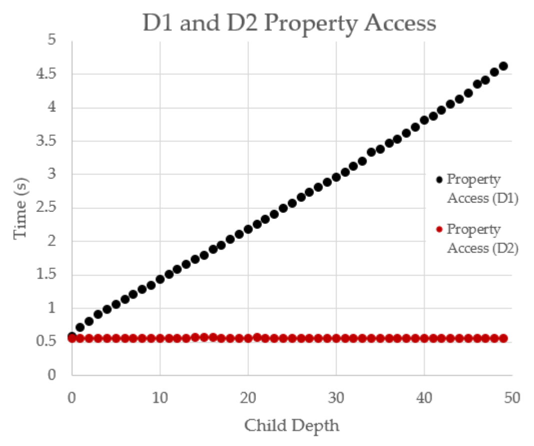

Figure 4: *A comparison of method call runtime between Design One and Design Two*

From this figure, it is apparent that we were correct in our expectations. The fact that each class owns its own reference to each inherited method means that the time required for method calls in Design Two remains constant even as the child depth increases. The additional work of creating a reference to each inherited method is done when the class is defined, which only happens once per class, and is not benchmarked by our testing framework. Our testing framework also does not measure the amount of memory taken up by our programs, and therefore, the only two drawbacks of Design Two do not manifest themselves in our benchmarks. For the remaining three tests of instance creation, data access, and metadata access, Design Two performs similarly to Design One.

ii. Design Three Results

From our discussion of our approach for Design Three, we expected the data access test to perform better in Design Three, but for instance creation test to suffer as a result.

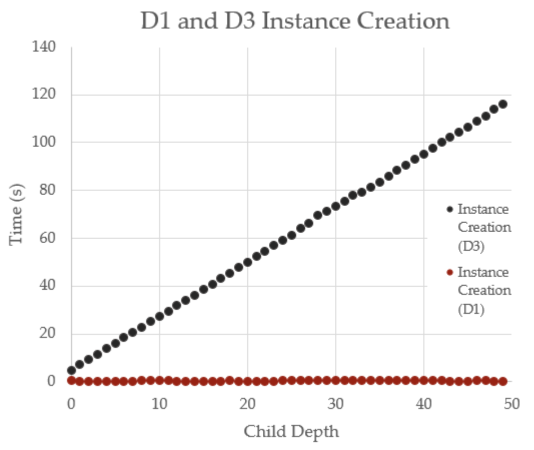

Figure 5: *A comparison of instance creation runtime between Design One and Design Three*

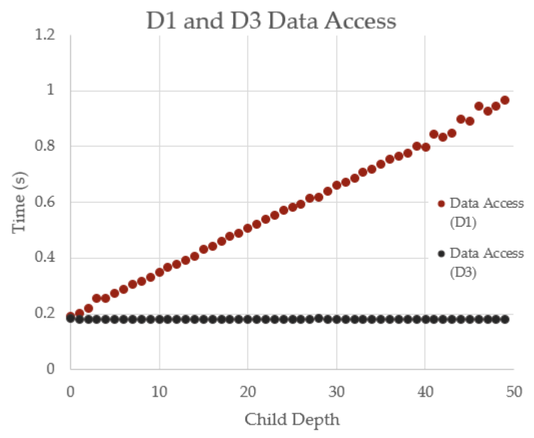

Figure 6: *A comparison of data access runtime between Design One and Design Three*

Indeed, as can be seen from the figures, going from linear to constant in child depth for data access caused us to go from constant to linear in child depth for instance creation. Instance creation was affected due to the possible need to call each ancestor's constructor. There is no way to definitely say whether Design One or Design Three is better, it all depends on the use case. For a use case where only a couple instances of the class are declared, but we access many data members of these instances, Design Three may be better. However, if we require many instances of the class, and will only rarely access the data members of each instance, Design One could have the advantage.

iii. Design Four Results

We expected Design Four to perform better than Design One, due to the use of the Lua C API and being able to define very precisely what we wanted to do. However, we did not know by what factor this improvement would be.

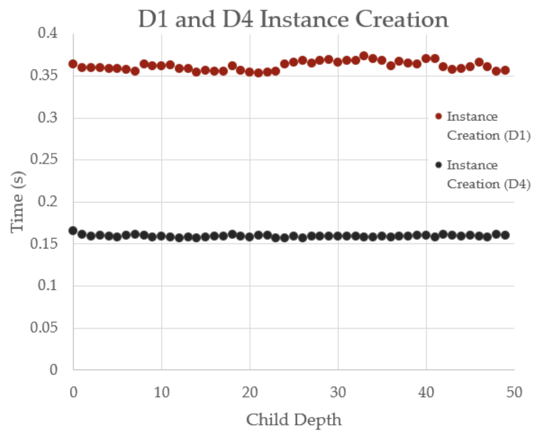

Figure 7: *A comparison of instance creation runtime between Design One and Design Four*

With the data from the figures, we can see now that there was an almost two-fold increase in the runtime of both instance creation as well as method calls when using the C implementation. The linear relationship between child depth and method call runtime is retained in Design Four since we still rely on metatables in order to find the proper functions to call. This improvement in runtime is also not as drastic as the improvement seen in Design Two; however, it also does not come at the cost of additional memory usage. For realistic use cases, classes will most likely have a child depth of less than ten. This puts Design Four in a sweet spot between Design One and Design Two, where it is approximately twice as fast as Design One, yet less than fifty percent slower than Design Two. In fact, as long as we are able to express our design of a class implementation using the Lua C API, there seems to be no reason not to convert it into a C implementation, especially for something that would be used as frequently as a class implementation.

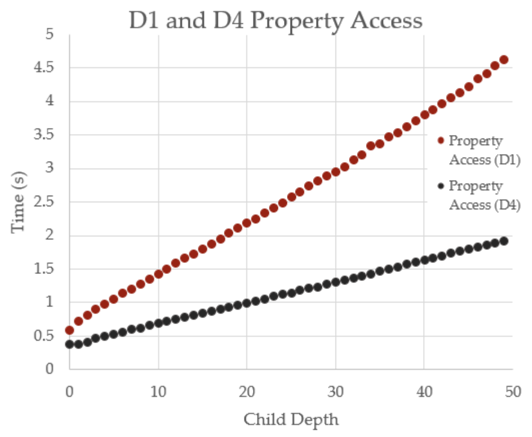

Figure 8: *A comparison of method call runtime between Design One and Design Four*

References

[Julien Patte, 2013] "Yet Another Class Implementation for Lua." https://github.com/jpatte/yaci.lua