Efficient Lua Class Implementation with Multiple Inheritance

NIHAT CEM INAN, Department of Electrical Engineering

1 SUMMARY

The purpose of this project was to implement an efficient class system in Lua and provide support for multiple inheritance in the class system. Since I already implemented a Lua class system for assignment two in CS242, I used that as a starting point [1]. Using a benchmark program, I measured the speed of various class operations performed by assignment two and discovered the areas that need improvement. It turned out that deep inheritance contributes to slowing down my class system operations. By studying how C++ handles the class system operations that could be faster in my program and using my own ideas, I improved the speed of these operations significantly. In addition to speed, I thoroughly tested the correctness of my class system operations and debugged my code so that all my class system operations behave correctly. It is important to note that my measurement unit for efficiency was speed. An interesting design tradeoff I discovered was that in order to improve speed in the class system, I had to use more memory. Afterward, I added multiple inheritance to my implementation which worked as I expected in my correctness tests for this feature. To implement multiple inheritance in a manageable manner, I restricted parent classes from passing duplicate items to child classes by designing the multiple inheritance feature such that items from the first parent class given to the argument list overwrite duplicates from other parents. Overall, my project was a success and coming from an Electrical Engineering background, this project provided me the opportunity to become more familiar with computer science concepts such as class implementation and multiple inheritance.

2 BACKGROUND

2.1 Necessary Background

Relevant background to understanding the context for my project includes being familiar with the Lua programming language, class system concepts, and the class system implementation section of assignment two [1]. My challenge was to make the class system in assignment two more efficient. Additionally, I wanted to add an additional feature to my class system that was not supported by the class system in assignment two, for which I chose multiple inheritance. Other features I could have chosen to support include abstract classes, interfaces, and sealed classes. Before completing this project, I was not very familiar with multiple inheritance, so I chose this feature to learn more about it. My project relates to this course's theme of expressiveness, since I implemented common programming constructs such as classes, variables, methods, and inheritance. My project also relates to the course's theme of performance, because I compared the performance of my efficient class system to the one I implemented in assignment two.

2.2 Review of Initial Class System from Assignment 2

Overview. To solve the challenge of designing an efficient class system with multiple inheritance, I needed to design a new system that performs class operations. Though I did not check substantially for efficiency in assignment two, the class system I designed for assignment two worked well in terms of accuracy. Therefore, instead of starting from scratch, I decided to start with the class system I built in assignment two and built on it. Before starting my work on efficiency and multiple inheritance, I decided to remove private variable support, since it is difficult to implement in Lua without introducing holes. Figure 1 illustrates some important ideas in my initial class system without the changes for efficiency and multiple inheritance. In the rest of this section, I discuss some details of this initial class system that are illustrated by Figure 1. This is the same class system implementation as the one for assignment two, so I assume that readers have familiarity with the class system specification provided in assignment two [1]. The only difference in the specification is that private variables are not supported.

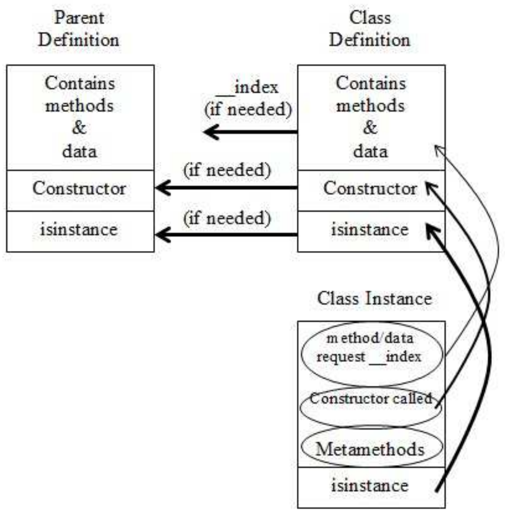

Fig. 1. Illustration of Initial Class System Operations.

2.2.1 Methods and Data. The Class.new() function creates a new instance of the class. The "__index" metamethod is set to lookup methods and data from the class definition when the instance needs them. The "__index" metamethods of the class method and data containers are set to look in the parent method and data containers. In other words, the "__index" metamethod of "Class.methods" is set to "Parent.methods" and the "__index" metamethod of "Class.data" is set to "Parent.data." Because parent classes use my class system as well, the "__index" metamethods of the parent class member contains are set to look in their parent member containers. As a result, when an instance of a class needs to use a class member, my implementation will look at the member containers of the class and all of its parents, and will return "nil" if nothing is found.

2.2.2 Metamethods. Whenever the instance of a class needs to access a method or data field, it uses the "__index" metamethod to access it using the class definition. So in reality, it is not the class instance that contains the methods and data fields but the class definition. Although metamethods of the class are contained in the class definition, they are also copied into each instance of the class. Unlike regular methods and data, all the metamethods of the class, including parent metamethods, are copied into the instance, so the lookup operations are not necessary; they can be accessed directly in the instance.

2.2.3 Isinstance. The class instance contains a method called "isinstance" which calls the "isinstance" function in the class definition. The "isinstance" function in the class definition returns true if the instance is an instance of that class, and calls the isinstance function of the parent otherwise. Since the parents use the same class system, their isinstance functions also either return true or call the isinstance function of their parent. This process continues until a match is found and isinstance returns true or no match is found and isinstance returns false.

2.2.4 Constructor. Since the Class.new() function creates a new instance of the class, it also calls the constructor to initialize the instance. The child constructor is called if it exists, otherwise, the parent constructor is called. Since the parent classes use the same class system, like the operations discussed before, if the parent class does not have a constructor, the constructor of the parent's parent will be called, and this pattern continues until a defined constructor is called.

3 APPROACH

3.1 Experiment Setup

3.1.1 Benchmark Tests. Though my final product was a system and not an experiment, I had to perform an experiment to determine the areas of my class system where operations could be faster. For the experiment, I had to create benchmark tests to evaluate the performance of the class operations. As a starting point, I used the benchmark tests from assignment two that were used to compare the performance of Lua method implementations versus C method implementations [1]. I made modifications to this assignment two benchmark test to more generically evaluate the performance of all aspects of my class system, such as method, data, and isinstance calls and how inheritance affects my class system performance.

3.1.2 Time Magnification of Class Operations. To make the inefficiencies clear to me, I isolated various class operations and magnified their effects. Isolated and magnified class operations include having cases for many methods, data, metamethods, many inheritance layers (deep inheritance). The setup controlled other variables by benchmarking these magnifications one at a time. In each case, the quantity of the other members is minimized. For example, the class with many methods does not contain many data fields, metamethods, or inheritance layers. Figure 2 below provides a visual representation of the experiment.

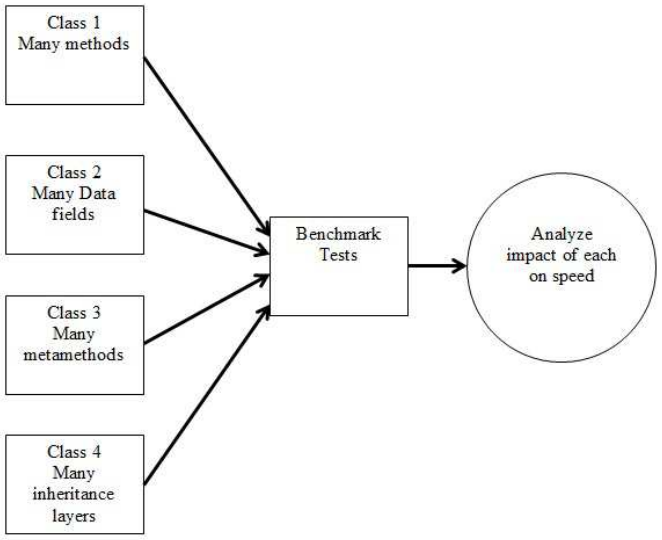

Fig. 2. Experiment Setup.

3.2 Experiment Results and Analysis

3.2.1 Experiment Results. The benchmark output below shows some of the output from running my benchmark tests. It turned out that having many methods, many data fields, and many metamethods, did not cause inefficiency. Deep inheritance however, as shown in the output below, caused a substantial amount of inefficiency in the cases where the last and deepest child class had to use a method or data field belonging to one of the first parent. For example, if a parent class has a child class, which has another child class, and this pattern continues until you have 100 inheritance layers, when the 100th child class uses a method or data field belonging to the first class (i.e. the first parent), the operation takes a significant amount of time. The "isinstance" feature of the class system was also inefficient in the deep inheritance case. Looking at the 100 layer example again, if you want to check that the 100th child class object is an instance of the first parent class, the operation also takes a significant amount of time. In the case below, there are two "isinstance." The first one, labeled "isinstance," checks if the deepest child class instance is an instance of itself. The second, labeled "isinstance(Object)" checks if the deepest child class instance is an instance of the object class, which is what comes first in the inheritance layers. The metamethods test, indicated by "Equality" below, and the constructor test, indicated by "Creation" below, did not have the efficiency issues of the other classes.

```
Benchmarking class: inheritance
===== Creation (1000000 iterations)  =====
Time:		2.54
===== Property Access (1000000 iterations)  =====
Time:		38.822
===== Arithmetic (1000000 iterations)  =====
Time:		84.179
===== Distance (1000000 iterations)  =====
Time:		38.889
===== Equality (1000000 iterations)  =====
Time:		0.51900000000001
===== To String (Lua) (1000000 iterations)  =====
Time:		40.004
===== isinstance (1000000 iterations)  =====
Time:		0.19300000000001
===== isinstance(Object) (1000000 iterations)  =====
Time:		58.164
```

3.2.2 Experiment Analysis. Without inheritance, method and data field call operations are fast because table lookup only occurs once where the "__index" metamethod looks up methods and data from the class definition. As more inheritance layers are added, the number of lookups increase. When you have a parent and child class, and need a method or data field from the parent for example, the "__index" metamethod looks up methods and data from the class definition, and then uses another "__index" metamethod from Class.methods or Class.data and looks up methods and data from the parent definition. If you have 100 layers, the lookup operation occurs 101 times! Each lookup operation takes time, and so in the 101 lookups case, the time it takes to execute a method or obtain a data field from the first parent can significantly increase, as shown in the figure below. The same problem occurs with the "isinstance(Object)" case, because as with the lookups in methods and data, the deepest child goes through many "isinstance" calls to reach the Object class, which adds time to the operation. Since parent metamethods are copied into the instance of the child class every time a new child class is created, access to metamethods occurs a lot quicker, hence the speed in the operation. A child class's constructor is also always in the child class definition, whether it is the child class's constructor or the parent class's constructor that is copied into the child class, which is why constructor operations occur quickly. These operations are similar to how C++ uses vtables, which will be discussed in the next section.

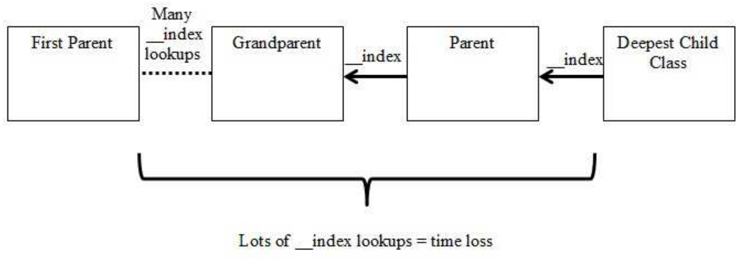

Fig. 3. Index time loss.

3.3 Inheritance Redesign for Methods, Data, and "Isinstance"

3.3.1 Vtables. It was suggested by one of the CS242 instructors, Varun Ramesh, for me to look into how C++ implements method lookup with vtables. In C++, when a child class is created, the child class contains a pointer that points to a virtual table containing pointers to class members [2]. The difference between my initial Lua class system design and the design of the C++ class system is that the virtual table for a child class in C++ contains pointers that directly point to the member definitions, regardless of if they belong to a parent class or child class. Therefore, you have direct access to all the members of the class, regardless of whether the members come from a parent or the child class. On the contrary, in the initial Lua class system design, with the exception of metamethods, direct access is only available for members of the child class and lookups need to be made for members coming from parents. The figure below shows the difference between the method and data lookups for C++ vtables vs. my initial Lua class system design.

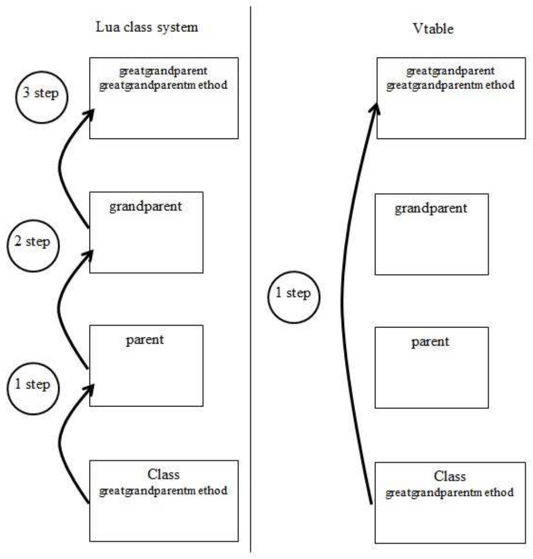
Lua class system
Vtable
greatgrandparent greatgrandparentmethod
3 step
grandparent
2 step
parent
1 step
Class greatgrandparentmethod
greatgrandparent greatgrandparentmethod
grandparent
1 step
parent
Class greatgrandparentmethod

Fig. 4. Vtable comparison.

3.3.2 Redesign for Method Calls and Data Access. My goal now at this point was to adopt the C++ vtable model of member lookup into my Lua class system. I had to implement the vtable model a bit differently since Lua does not support explicit pointers like C++. Since my implementation for metamethod lookup was similar to the vtable model, I decided to implement my method and data lookup systems in a manner similar but not identical to my metamethod lookup system. Specifically, when a new class definition is created, all the methods and data fields of the parent classes are copied into the new class definition. This is different from my metamethod lookup system because parent metamethods are copied into child metamethods in each instance of the child class, whereas regular methods and data fields are copied into the child class only in the child class definition. The metamethod "__index" is still used by the instance of the child class to look into the child class definition, like my initial class system design. The difference now though, is that the child class definition contains all the methods and data fields of the child class, including the ones inherited from parents, and therefore, additional lookups with the "__index" metamethods of Class.methods and Class.data are not necessary, unlike the initial class design. It is also important to note that when I say "copied methods," I mean that references to the method definitions are copied. Therefore, all the copies of method references share the method definition, making it similar to the "pointer structure" in the vtable model and sharing the same method definition saves memory space. I anticipated at this point that this redesign for method lookup and data access systems would operate significantly faster than the ones in my initial class system. My method call system for instance would look similar to the vtable model in figure 4 above.

3.3.3 Redesign for Isinstance. My redesign for "isinstance" was different than my method call and data access redesigns, but still influenced by the vtable model. I assign every class definition an ID, with the base for all Lua classes "Object" starting from an ID of 0. Every time a class definition is created using my class system, a new ID is given to the class that is one greater than the previous ID given to another class definition. Note that the ID numbers do not have to follow a pattern, as long as they are unique. In a class definition, the current class's ID, as well as all of its parents' IDs are stored in an "instance table" as keys, and their values are all assigned to 1. The 1 is just an arbitrary number I chose to indicate that the keys exist in the table. So, if a key exists, it has a value of 1, otherwise, it has a value of nil. When the user uses the "isinstance" method, the method returns true if the key, which is the ID of of the class, has a value of 1 in the "instance table," and false if the value is nil. This implementation of "isinstance" is more efficient than the one in the initial class system, because, as repeated parent lookups were eliminated in the method call and data access redesigns, repeated parent "isinstance" calls are eliminated in the new "isinstance" redesign. Since the IDs of the class and all of its parents are stored in the "instance table," if you want to check if the class is an instance of for example Class A, you return true if the value assigned to the key (ID of Class A) is 1 and false if it is nil, and so the operation is completed with a single lookup in all cases. The figure below illustrates this concept.

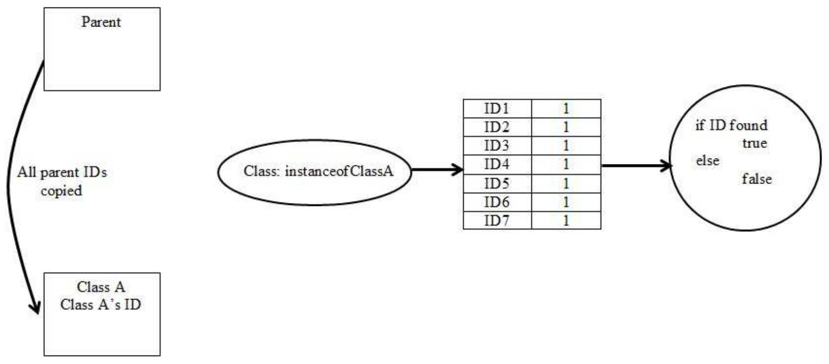

Fig. 5. Instance table and "isinstance" operation.

3.4 Addition of Multiple Inheritance

3.3.1 Research. Since I never dealt with multiple inheritance before until this project, I spent some time researching what multiple inheritance was and what challenges there are in implementing it. I found out that multiple inheritance is when a child class inherits from multiple parent classes at the same inheritance level [3]. I found out that I needed to watch for problems like diamond inheritance, and more generally, I needed to figure out how to handle duplicate parent class members when they are passed to the child class [3].

3.3.2 Initial Ideas on Passing Parents to Child. Before dealing with issues like diamond inheritance, I had to first figure out how to implement multiple inheritance in general in Lua. A challenge of implementing multiple inheritance in Lua was that I needed to simultaneously support single inheritance as well. More generally, a class had to be able to inherit from a variable number of parents. An initial idea I had was that I could overload my class system definition and when the user defines a class with an arbitrary number of parents, the overloading would handle the variable number of parents. However, I found out that overloading is not supported in Lua. Plus, I would need a class system definition for every combination of arguments I want to support which would be a lot of work. Also, I did not want to put limits on how many parents can be supported.

3.3.3 Use of Tables for Passing Parents to Child. After brainstorming ideas, I decided that passing in a table of parents into the class system definition would be the best way to pass parents classes to the child class. This way, I do not need to overload my class system definition and the table size can vary. Using a table of parents did require me to slightly change the way single inheritance would be used. Instead of passing a parent class as an argument like in assignment two, the user had to pass a parent table with one element. I accepted this change since it was only a minor inconvenience compared to the substantial benefits I gained by passing in a table of parents to the child.

3.3.4 Vtable Model Applied to Multiple Inheritance. I was able to apply the vtable model to multiple inheritance as well. When a user creates a new class definition, all the methods and data fields from all the parent classes get copied into the new class definition. This design worked well in many cases, but I needed to figure out how to handle duplicate members.

3.3.5 Brainstorming for Handling Duplicate Members. I spent some time brainstorming how to handle duplicate members of parent classes and came up with an algorithm that could potentially be implemented. Each time a member from the parent class is copied into the child class, the parent of that member is copied into a “parent table” which includes the member names as keys and the parent names as values. When a duplicate member is detected, the program can lookup which parent the member originally came from and make two versions of that member where the name of the member would turn into the original member name concatenated with the parent name. For example, if you have an “add” method coming from Class A and an “add” method coming from Class B, the names in the child class would turn into “addClassA” and “addClassB.” As a useful feature, the definition of the original member, which is in this example “add,” would be replaced by a new definition that contains a print statement telling you the choices you have for that member. So, in this example, “add” would print the following:

```
Please select an add function.
addClassA
addClassB
```

This is similar to the way C++ keeps track of which parents members came from. Though this algorithm performs well in many cases, it does have issues. For instance, in the “add” example from above, hypothetically, what if there was another method in the child class that was already called “addClassA”? Then you would have to call your “add” method coming from Class A something else. What if the method’s new name also already exists? One solution to this issue would be to add restrictions on what member names can be, but I preferred not to do this because many programming languages offer support for flexible member names, and restrictions for flexible member names would be difficult to keep track of. C++ contains syntax specifically implemented for showing which parents members come from. Because it was not clear to me how to add this type of syntax to my Lua class system, I decided to not use the algorithm I described for handling duplicate members.

3.3.6 Decision for Restriction. I decided to solve the duplicate member issue by adding a restriction. I decided that if there are duplicate members, the members of the first parent in the table of parents to copy its members to the child would override duplicate members of the other parents of the table. The “table.merge” function I use to copy class members makes the implementation of this design convenient because the “merge” operation behaves in the same manner as my design. Specifically, if a child class already contains a value for the member key that is not nil, it does not let any other parent override that value. Since I copy class members from parents in order of increasing parent indices, the members of the parent at index 1 for example would override duplicate members of the parent at index 2. Members of the parent at index 2 would override duplicates from the parent at index 4. The restriction allows for a clearly defined behavior of the class system and does not take away much from the class system features. If the user wants to pass duplicate members to a child class, they can simply change the names a bit and pass them.

4 RESULTS AND DISCUSSION

4.1 Results

In terms of my design evaluation, my class implementation should be able to perform the class system mechanisms accurately. It should also be evaluated in terms of efficiency, where my class system should be able to demonstrate that various class system operations are faster than before. In terms of accuracy, including my addition of multiple inheritance, my class system passed the tests provided in the “class_tests.lua” file (with the exception of private variable tests since private data support is removed) plus other additional tests, a summary of which is provided below:

1. Accessing a class data field directly, since private variables are no longer supported
    a. Child class instance accessing its data field directly
    b. Child class instance accessing its inherited parent data field directly
    c. Child class instance accessing its inherited grandparent data field directly
2. Method call
    a. Class instance calling its own methods, parent methods and grandparent methods
    b. Parent from “a.” calling a method from its parent (the grandparent in “a”)
3. Constructor Calls
    a. Child class calls its own constructor when it exists, and the parent constructor otherwise. Unlike the “class_tests.lua” tests, I make the constructor implementations different so I can make sure the correct constructors are being called at the correct times.
4. Isinstance
    a. Check if a class is an instance of another class that does not have any inheritance connection to the first class. This should turn out to be false.
5. Nil and “Error” tests

   a. Class instance calling a method from another class that has no inheritance connection to the first class. Should return nil.
   b. Class instance accessing a data field from another class that has no inheritance connection to the first class. Should return nil.

Tests that involve Multiple Inheritance

1. Make sure all the single inheritance tests from “class_tests.lua” pass with multiple inheritance modifications.
2. Modify the grandchild class in “class_tests.lua” so that it inherits from the child, as well as another parent, called parent2. Create a new instance of the grandchild class and perform isinstance tests:
   a. Grandchild instance is an instance of class.Object, returns true
   b. Grandchild instance is an instance of parent class, returns true
   c. Grandchild instance is an instance of child class, returns true
   d. Grandchild instance is an instance of grandchild class, returns true
   e. Grandchild instance is an instance of parent2, returns true
   f. Grandchild instance is an instance of parent3, returns false, since parent3 does not have any inhertiance connection to Grandchild
3. Create a class that inherits from Parent1 and Parent2
   a. Class instance should be able to call its own methods, Parent1’s methods, and Parent2’s methods
   b. Class instance should be able to access its own data fields, Parent1’s data fields, and Parent2’s data fields.
   c. Class instance should call Parent1’s constructor, since it is first parent (it is the parent at index 1 in the parent table)
4. Create a class that inherits from Parent1 and Parent2, where Parent1 and Parent2 contain duplicate members. Parent 1 appears first in the parent array.
   a. Make sure Parent1 members are the ones the class uses.

Considering the thoroughness of the tests above, my class system executes its operations accuractely. In terms of efficiency, the benchmark comparisons for the operations I tried to improve are shown below:

Initial Lua Class System Implementation	Final Lua Class System Implementation
Benchmarking class: inheritance ===== Property Access (1000000 iterations) ===== Time: 38.822	Benchmarking class: inheritance ===== Property Access (1000000 iterations) ===== Time: 0.635

===== Arithmetic (1000000 iterations) =====	===== Arithmetic (1000000 iterations) =====
Time: 84.179	Time: 3.512
===== Distance (1000000 iterations) =====	===== Distance (1000000 iterations) =====
Time: 38.889	Time: 0.674
===== To String (Lua) (1000000 iterations) =====	===== To String (Lua) (1000000 iterations) =====
Time: 40.004	Time: 1.533
===== isinstance(Object) (1000000 iterations) =====	===== isinstance(Object) (1000000 iterations) =====
Time: 58.164	Time: 0.268

As shown above, the data field access operation with deep inheritance, demonstrated by “Property Access,” is much faster in the final class system that it is in the initial class system. The same is true for method calls with deep inheritance that include arithmetic, distance, and ToString. For “isinstance,” checking if the “deepest” child is an instance of class.Object is much faster with the new class system design than it is for the initial class system design. The comparison is fair because the same benchmark tests are being run on the class systems and the class systems are performing the same operations. The only variable that is different in the tests is the class system. The results demonstrate that I was successful in reaching my goals, because I significantly improved the speed of class system operations that were initially slow.

I also benchmarked my class system in LuaJIT, which contains a just-in-time compiler for Lua [4]. Since it has C implementations internally, it runs faster than Lua without the JIT compiler[4]. Below is a summary of my findings in LuaJIT. It is important to note that my LuaJIT comparison was done on the Stanford rice machine, whereas all other benchmark tests were done on my local windows computer. Therefore, the times shown in the LuaJIT comparison below are not to scale with the previous benchmark results.

Fianl Lua Class System Implementation with Lua	Final Lua Class System Implementation with LuaJIT
Benchmarking class: inheritance	Benchmarking class: inheritance
===== Creation (1000000 iterations) =====	===== Creation (1000000 iterations) =====
Time: 1.297744	Time: 0.51053
===== Data Access (1000000 iterations) =====	===== Data Access (1000000 iterations) =====
Time: 0.173905	Time: 0.029454000000001
===== Property Access (1000000 iterations) =====	===== Property Access (1000000 iterations) =====

Time: 0.934028	Time: 0.341602
===== Arithmetic (1000000 iterations) =====	===== Arithmetic (1000000 iterations) =====
Time: 4.483167	Time: 1.841963
===== Distance (1000000 iterations) =====	===== Distance (1000000 iterations) =====
Time: 0.977152	Time: 0.085206999999999
===== Equality (1000000 iterations) =====	===== Equality (1000000 iterations) =====
Time: 0.707791	Time: 0.106775
===== To String (Lua) (1000000 iterations) =====	===== To String (Lua) (1000000 iterations) =====
Time: 2.410879	Time: 1.206889
===== isinstance (1000000 iterations) =====	===== isinstance (1000000 iterations) =====
Time: 0.375715	Time: 0.147947
===== isinstance(Object) (1000000 iterations) =====	===== isinstance(Object) (1000000 iterations) =====
Time: 0.412786	Time: 0.163962

4.2 Discussion

4.2.1 Analysis of Results. My final class system design performed faster than my initial class system design in the benchmark tests because, as described in the "Approach" section of my report, I reduced the number of "lookups" the class system has to go through. Though the implementations for each class system operation I improved have their differences, all of the implementations are based on the concept of reducing lookups while going through inheritance layers. In the new implementations of method call and data access, all the parent methods and data exist in the class definition, allowing the class to access them directly, rather than going through inheritance layers. In the case of "isinstance," all of the class IDs are stored in an "instance" table so that the class definition can perform the "isinstance" operation by looking up the ID of the class the user wants to perform an instance check with, rather than going through the inheritance layers until a match is found.

4.2.2 Memory Tradeoff. Though I increased the efficiency of my class system, there are some tradeoffs I had to make. First of all, I am using more memory since each time a child class is created all the parent class members are copied into the child class. The system uses more memory to store these copies. With the initial class system implementation that used many lookups in its operations, the number of copies stored was significantly less. The initial class system often reused the same member definition for all classes that had access to the member, though it took some classes longer to access the member because of the inheritance layers. From my perspective, since the extra memory space was available, I believe that it is worth it to use some extra memory in order to increase the speed of some of my class system operations.

4.2.3 Start Speed Tradeoff. In terms of speed, I do lose a bit of speed when I create my class definitions, since each time I create a class definition, I copy the methods and data fields of the parent class into the child class. In my initial class definition, I would just set the "__index" metamethods of Class.methods and Class.data to Parent.methods and Parent.data respectively, which took less time than copying methods and data. My implementation may add time to the startup of the program because it may take longer to load the classes. However, instances of classes are typically created more often than class definitions. For example, a single class definition could be instantiated 5 times. Though some time is lost in the creation of the class definition, lots of time is gained in the operations that are performed by the class instances. Therefore, I believe it is worth it to make this trade.

4.2.4 Runtime Impact of Multiple Inheritance. Lastly, I would like to discuss the runtime impact of multiple inheritance. Since the initial class system implementation did not support multiple inheritance, I could not make a comparison to it in terms of efficiency. Still, I wrote a class file that made use of multiple inheritance to a reasonable extent, and the benchmark results I got for method calls, data access, and isinstance did not change very much. This is because in terms of the operations measured in the benchmark tests, the only difference multiple inheritance makes is that more parent class members are copied into the child class. Like single inheritance, all the multiple parent methods and data exist in the class definition, allowing the class to access them directly and again there is no need to go through any inheritance layers. In the case of "isinstance," all the IDs of the multiple parents are stored in the child's "instance table," allowing them to be accessed directly as well. From the perspective of the child class, multiple inheritance really only means more members get copied into the child class. This being said, like my single inheritance implementation, my multiple inheritance implementation could potentially add more time to the loading of the class definitions, because having multiple inheritance may often result in having more members to copy to the child class, which could increase the startup time of the program, as discussed before. However, instances perform class system operations quite efficiently, and since class instances are typically created more than class definitions, it is worth it to lose some time with the class definitions and gain lots of time when class instances perform class system operations.:

5 CONCLUSIONS

In Overall, my project was a success. Likely because I come from an Electrical Engineering background, some of the computer science concepts I dealt with in this project such as multiple inheritance were new to me, and so this project provided me the opportunity to become more familiar with these concepts. I hope that I can work on more class system projects in the future and apply what I learned from this CS242 project to those projects.

REFERENCES

[1] Will Crichton, Varun Ramesh, Jintian Liang, and John Clow. 2017. CS 242: Programming Languages (Fall 2017). Retrieved December 14, 2017 from http://cs242.stanford.edu

[2] Learncpp.com. 2017. Learncpp.com: Tutorials to help you master C++ and object-oriented programming. Retrieved from http://www.learncpp.com/

[3] Alex Allain, Alex Hoffer, Michael Kern. 2017. C Programming.com: Your resource for C and C++ (2017). Retrieved December 14, 2017 from https://www.cprogramming.com/

[4] Mike Pall. 2017. The LuaJIT Project (2017). Retrieved December 14, 2017 from http://luajit.org/index.html