EWG, SG14: P0130R0
27-9-2015
Scott Wardle, Roberto Parolin
swardle@gmail.com, robertoparolin@gmail.com

Comparing Virtual Methods

I. Summary

It would be useful to compare if an instance of a class has a particular function in its virtual table.

II. Motivation

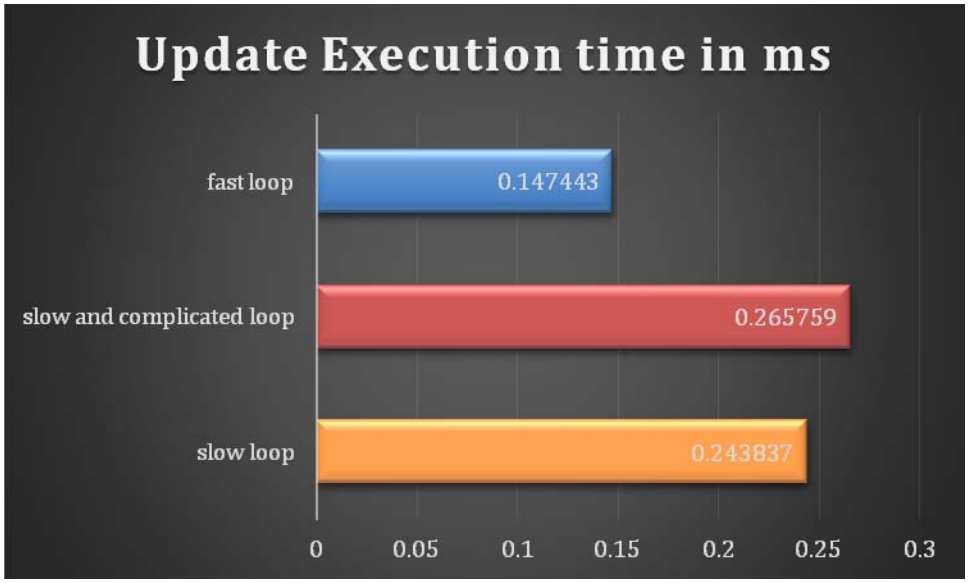
Traditional entity systems in c++ are used often in games. An example of one of these entity systems would be if you have a list of entity base classes some of which are overridden and not known at compile time and are data driven. You would then update these entities by looping over a vector of pointers to entity base class and calling each entity update function.

We could separate each entities based on the update function in the virtual table moving these overridden objects out of the fast path and have one function that updated many non-overridden entities. Then update each overridden entity slowly afterwards.

To show the power of this idea. I created 3 loops see below for source code. (The full source is in cpp_entity_example\ *.cpp *.h)

1. Slow loop
    a. Calling a virtual function for each loop
2. Slow and complicated loop
    a. Trying to avoid calling “expensive” function with GetType virtual function.
3. Fast loop
    a. Trying to avoid calling “expensive” function with pointer to a 64 bit int like what would happen if I could compare to a method in the virtual table.

The function in my example are not very expensive so I can see virtual function call overhead clearly but here are the numbers in my example.



	ms	%
slow loop	0.243837	92%
slow and complicated loop	0.265759	100%
fast loop	0.147443	55%

1 slow loop

```
mytimer timer;
for (float t = 0.0f; t < 1.0; t += 0.05f) {
	for (auto &a : entity_vec) {
		a->Update(t);
	}
}
gSlowSimpleUpdateExampleTimers.emplace_back(timer.stop());
```

2 slow and complicated loop

```
mytimer timer;
for (float t = 0.0f; t < 1.0; t += 0.05f) {
	for (auto &a : entity_vec) {
		if (a->GetType() != entity_lerp_fast::type)	{
			a->Update(t);
		}
	}
	entity_lerp_fast::UpdateAll(t);
}
```

```
gSlowComplicatedUpdateExampleTimers.emplace_back(timer.stop());
```

3 fast loop

```
mytimer timer;
for (float t = 0.0f; t < 1.0; t += 0.05f) {
	for (auto &a : entity_vec) {
		if (*a->m_typedata != entity_lerp_fast::type) {
			a->Update(t);
		}
	}
	entity_lerp_fast::UpdateAll(t);
}
gFastUpdateExampleTimers.emplace_back(timer.stop());
```

So why not just add the type data like in the fast example? The main problem is this is nearly always a refactor job. First we write things simple like the first loop. Then only once we see the performance problem do we slowly start moving towards the fast loop. So the less lines of code I can change and get the performance the better.

Entity system are very important structure in games. One many games you will have up to 300 content people and only 30 programmers. The content people will make entities and components for these entities. The programmers will have little say in how many of any entity are used. They will see what type of prototype scene content people are trying to make and then try to update their game code to make these scenes faster as needed. But they should try and not change the engine itself or at least make sure the changes are small. The idea is to only make these type of changes where it was worth it to get the cycles back otherwise we would just stick to simple virtual functions.

For entities you can think of them as things that do stuff in the world and components as the tools entities can use. Entity could be very complicated like a car in a race game or a soldier in battle field or could be very simple like a tree. You might want to support lot and lots of simple objects like trees and they could be updated very quickly and uniformly with SIMD matrix and vector math. With large objects like soldier or race car these virtual function will not mater. The nodes in skeleton animation hierarchy in a soldier entity have a similar problems to the entity one I talk about mind you so a sports game that has more animating players then battle field might apply this tech there to more effect. The entity problem is just one easy to understand where we would use it.

In some cases we could sort the entity based on type and update them. This would give us good saving in these cases as we would get less branch misprediction however this is not always posable. Even in these cases comparing data and not calling a virtual function would be better.

A static typing systems could be writing in some cases but the entity hierarchy is a good example of a hard case. Since each main player is quite different soldier verses rocket ship each team will write their own entities so the whole system will be 100s

of files over 20 development teams. So the changes cause integrations problems when we have to change the type system.

So the basic problem with type systems is you don't know you need a better one until you see the need for optimization. You don't know that you need optimization until you see you scene. Once you have your scene you written a lot of code the wrong way and you need to refactor it to make it faster to make the scene look good.

III. Additional work

Sorting by using the virtual table it's self not function as typeid, it might be useful in some cases (remember we don't use rtti). However this is a separate problem. But maybe the destructor function could use used as a typeid for the class as there can be only one of these even with DLLs unless you inline the destructor.

Are the function in vtables always non-stubs for calling across DLL boundaries? It would be better if methods would only have one address. If stubs were needed for long jumps or crossing DLL boundaries would this cause trouble. This might add one more level of indirection or add more data to the virtual table but I don't see this as being a show stopper at this point.

III. Discussion

Roughly I want to do things like the following C# code can.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace cs_func_sort
{
   class A
   {
      virtual public void Foo()
      {
         Console.WriteLine("A::Foo");
      }
      virtual public void Name()
      {
         Console.Write("Class A ");
```

```
    }
}
class B : A
{
    override public void Foo()
    {
        Console.WriteLine("B::Foo");
    }
    override public void Name()
    {
        Console.Write("Class B ");
    }
}

class C : B
{
    override public void Name()
    {
        Console.Write("Class C ");
    }
}

delegate void fptrDelegate();
class Program
{
    static void Main(string[] args)
    {
        List<A> list = new List<A>();
        list.Add(new A());
        list.Add(new C());
        list.Add(new B());
        list.Add(new A());
        list.Add(new C());

        Console.WriteLine("Print the list unsorted");
        foreach (A cur in list)
        {
            cur.Name();
            cur.Foo();
        }


        Console.WriteLine("Print the list sorted");
        list.Sort(delegate (A itemA, A itemB)
        {
            fptrDelegate afoo = itemA.Foo;
```

```
            fptrDelegate bfoo = itemB.Foo;
            return afoo.Method.GetHashCode() - bfoo.Method.GetHashCode();
        });

        foreach (A cur in list)
        {
            cur.Name();
            cur.Foo();
        }

        Console.WriteLine("Find all of the version with B");
        // Can't think of a good way to get rid of creating this object in C# in C++
        // in C++ it would be better if we didn't have to create a instance of the class
        //  to make a pointer to function like the example below in yellow.

        A testA = new B();
        fptrDelegate testfoo = testA.Foo;
        foreach (A cur in list)
        {
            fptrDelegate curfoo = cur.Foo;
            if( testfoo.Method.Equals(curfoo.Method) )
            {
                // call B::Foo AND C::Foo!!
                cur.Name();
                cur.Foo();
            }
        }
    }
  }
}
```