Audience: EWG, SG14: D0130R1  
Date: 28-02-2016

Reply to: Scott Wardle, Roberto Parolin, Paul Hampson, and Sunil Srivastava,  
[swardle@gmail.com](mailto:swardle@gmail.com), [robertoparolin@gmail.com](mailto:robertoparolin@gmail.com)

**Comparing Virtual Methods**

# I. Summary

When refactoring entity systems for speed it can be useful to compare if some object has a method or not. A way if a instance of a class has a particular function in its virtual table would be useful. Currently we can do this using the GCC extension -Wno-pmf-conversions and achieve a more localized change to the code with similar performance to say using enums and better performance then using a extra virtual function call.

# II. Motivation

Entity systems written in C++ are used often used in games. A simple (and maybe naïve) entity would be a class with a virtual update function. Simple (and maybe naïve) entity systems could be a vector of Entity base pointers. The main loop would call each entities update function once per frame by looping though each entity and calling update.

Once you had this type of entity system, programmers (or more likely content creators) would then add their own entity like soldiers, zombies or basketball players to the list of entities and you would update the whole system of basketball players, zombies and soldiers just by calling the entity system update. Programmers setup the entity type and content creators would make the entity part of the world.

Each game wants to have new types of entities. A basketball player does not move like a zombie, so the entity behavior must differ. Game teams don’t want to handle all problems involved with making entities that walk because these are very hard problems. For example, you need a whole animation system to make walk cycles, etc. Games share large libraries of functions between each other and this is typically called an engine.

Engine programmers might write walking entities for a first person zombies and soldier game. A basketball player however needs a higher quality walk animations because you are focused on how the player moves and spend the entire game looking directly at him.

This story explains the 2 types of objects that are in entity systems.

* The “in engine” entity like the zombie and soldier
* The “game team overridden” entity for a basketball player.

Because the game team made the basketball player as an entity, they can add zombies to their basketball game if they choose.

The engine programming team does not know about the basketball player (because that is a game specific entity) but wants to make zombies faster because everyone likes to have lots of zombies in their games. What they would like to do is take a group of “walking entities” and make a zombies group update function. The basketball player entity that engine team didn’t write (and isn’t aware of) might still want to use the old walking entity functionality. This is where comparing virtual functions comes in.

We could separate each entity type based on the update function in the virtual table. Moving the overridden objects out of the fast path and have one function that updates many non-overridden entities. Then update each overridden entity slowly after-wards.

Here are some advantages:

* No new data is necessary
* Decreases i-cache misses
* Improve branch prediction, if statements can be predicted on more hardware then virtual calls.
* Allow grouping entity objects even if we started out with poor maybe naïve design.
* Empowers the programmer to reuse vtable as data not just functions.
  + Do more with the same data.
* Lets the user make a sort key that is as complicated as an objects vtable or as simple as one element of it.

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. **1 virtual function (slow loop)**
   1. Calling a virtual function once per-entity per-iteration
2. **2 virtual function (slow loop and complicated)**
   1. Trying to avoid calling “expensive” functions with GetType style virtual function. Then calling the original once per-entity per-iteration for all of the slow update entity.
   2. Finally use a fast-optimized loop to update a group of entity with no virtual functions.
3. **Enum (Fast loop)**
   1. Use a type enum to avoid the calling GetType virtual function.
4. **Member Function Compare (Fast loop)**
   1. Use the GCC extension -Wno-pmf-conversions to compare for a function in the virtual table.

I tested each of these on gcc 4.8.4 on ubuntu 14.04.1 virtual box on my 11” air 2012 and got similar results on my pc also on virtual box.

Note: The functions in my example are not very expensive so I can see virtual function call overhead clearly. As the function over head of the virtual function go down with the size of the real update function the differences would be smaller.

Speed Test Chart 1

## The source code for the test

Here is the 4 loops that were tested.

## One Virtual Function

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());

## Two Virtual Functions

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());

## Enums

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());

## Member Functions Compare Using GCC extension -Wno-pmf-conversions

typedef void (entity::\*memfun)(float y) const;

memfun mf = &entity::Update;

as\_normfun snf = (as\_normfun)(&entity\_lerp\_fast\_impl::Update);

for (float t = 0.0f; t < 1.0; t += 0.05f) {

for (auto &a : entity\_vec) {

const entity& e = \*(&(\*a));

as\_normfun dnf = (as\_normfun)(e.\*mf);

if (snf != dnf) {

a->Update(t);

}

}

entity\_lerp\_fast::UpdateAll(t);

}

Why not add an enum like the one used in the fast example? The main problem is this can be a large refactor job in our codebases. The simplest code is the one virtual function version. Once you see this as a hot spot in profilers you will check code around that hot spot while avoiding making changes in every single class if possible.

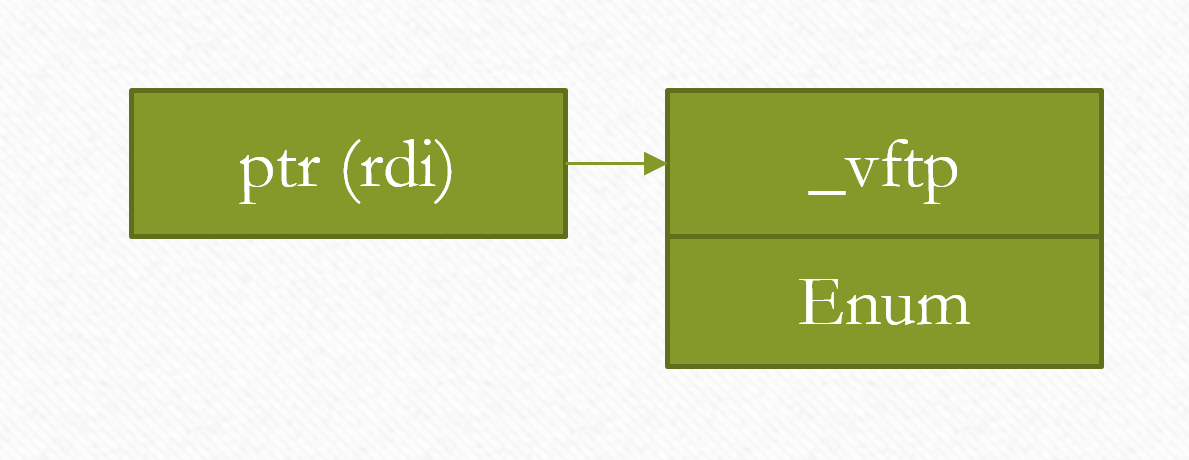
The ASM output of the Enum version vs Member Function Compare is very close.

ASM listing 1 Enum vs Member Functions Compare

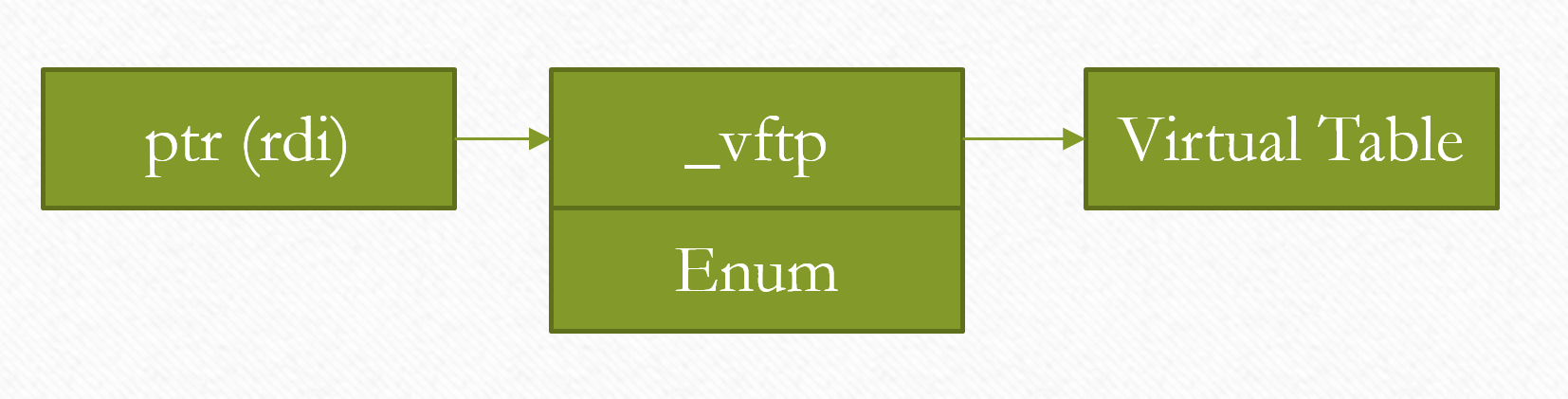
|  |  |
| --- | --- |
| Enum | Member Functions Compare |
| if (\*a->m\_typedata != entity\_lerp\_fast::type) {  mov 0x8(%rdi),%rax  cmp %r12,(%rax)  je 404417 <\_Z23FastUpdateExampleTimersv+0x4d7>  a->Update(t);  mov (%rdi),%rax  movss 0xc(%rsp),%xmm0  callq \*(%rax) | const entity& e = \*(&(\*a));  as\_normfun dnf = (as\_normfun)(e.\*mf);  mov (%rdi),%rax  mov (%rax),%rdx  if (snf != dnf) {  cmp $0x402c80,%rdx  je 403e0a <\_Z32MethodPointerUpdateExampleTimersv+0x4ca>  a->Update(t);  movss 0xc(%rsp),%xmm0  callq \*%rdx |

As expected the main difference is the enum is fetched from within the class data while the member function gets data that is stored inside the virtual function table. See the 2 diagrams below.

**Enum structure diagram 1**



**Member Functions Compare Structure Enum structure diagram 2**



The extra level of indirection is a mixed blessing. It is one more pointer to chase, however compared to chasing the this pointer then calling a function. The variation that compares the function pointer itself to a value is cheaper.

Entity systems are very important structures in games. Many games will have up to 300 content people and only 30 programmers. The content people will make components for these entities in the engine provided editor. The programmers will have little say in how many of the entities are used. They typically only see what prototype levels content people are trying to make and then try to optimize 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 types of changes where it was worth it to get the cycles back otherwise we would just stick to simple virtual functions.

Game engine entities are things that do stuff in the game world and entity components as the tools entities can use. Entities can be very complicated like a car in a race game, 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 soliders skeleton animation hierarchy have a similar problems to the entity ones I talked about. Sports games have more animating players then some games that take place on a battlefield so applying this tech there could have more effect. The entity problem outlined is just one such area that is easy to communicate and understand how we would apply this solution.

In some cases we could sort the entities based on type and update them. This would produce good savings in these cases as we would get less branch mispredictions however this is not always possible. Even in these cases comparing data and not calling a virtual function would be better.

A statically typed system could be written in some cases but the entity hierarchy is a good example of a hard case. The main player entity is quite different in individual games (eg. a soldier verses a rocket ship) each team will write their own game entities so the whole system will be 100s of files over 20 development teams. So any specific tuning an individual game team makes to common engine code causes integrations problems across the company when we have to change the types.

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 a prototype scene. Once you have your scene, you wrote a lot of code the wrong way and you need to refactor it to make it faster to make the scene look good.

# III Issues with this system

One interesting problem is comdat folding that some linkers do. Optimized code with comdat folding means simple function like the 2 below would become the same function in the linker and get the same address.

size\_t foo () { 0 return;}

void \*bar() { 0 return;}

You would have to be very careful. It does reduce the number of cases that this feature is useful for example a stub destructor would be the same address as all other stub destructors. A workaround for this problem would be very helpful. Most ideas I have are painful and compiler specific: like adding inline asm that pretends to access a function static var. I would still want this feature even if a perfect solution can’t be found but that would mean it could only be used by “experts”.

Some comments from **Sunil Srivastava in black with red comments from Scott Wardle.**

On the positive side, there already is an implementation; in GCC, for quite some time.

On the negative side:

a) It does not always work. You can not guarantee a unique pointer to the function across all possible vtable implementations having thunks in multiple files. If two pointers obtained this way compare equal, you are guaranteed that calls to them will go to same place. If they do not compare equal, the calls to them may still go to same place. It looks like that may be acceptable for this particular purpose, but I am not sure.

Like you I don’t think this is a problem for the given use case. If it is not a speed gain I would not use it. But we should look at the unintended consequence I am starting to see more with your advice. We should write some tests to prove when thunks are created. Does anyone have a good idea where to start?

For example on PPC I saw them when calling over 24 bits away in my exe/elf. But this would not have happened with virtual function as they would have had a 32 bit offset so they could address all of memory. But this is a very rare side case that does not seem to apply.

b) The performance may be questionable, because the conversion (in the link above)

fptr p = (fptr)(a.\*fp);

involves a test to check whether fp points to a virtual-member-function or a non-virtual-function. That test is futile for us because we are doing this conversion on virtual-member-functions only. This test will be done on every object. Perhaps we can figure out a way to avoid this test. The performance bar for this feature has to be fairly high in order for this to be useful.

I don’t see this problem in the ASM generated from gcc 4.8.4 -O3 anyways. Maybe this is a very good implementation. I don’t see the compare you are talking about for example. I think this should be a very easy thing for a complier to work out statically is a function is non-virtual or not. It should be the same rules for calling the function.

The being said addressing the right function statically if you had multiple inheritance I have not tried to do this yet. You need 2 function pointers in my test. One a statically bound one that I get before the loop and a dynamically bound one for each object and compare these together. I do worry that the static one might be a tricky to address in multiple inheritance cases. This would stop this code from being as useful in my real test.

c) It smells like the second half of the ill-fated unified-call-syntax (n4474). It converts a member function to plain function, though, as Ville pointed out, the unified-call-syntax does not provide this capability.

I see what you mean it does have the unintended consequence of making function pointer (but not normal functions) have the unified-call-syntax. I don’t see how to get around this issue. Maybe this is a feature not a bug.

Maybe one solution would be to add some type of special cast as you need to cast to create either of the static and dynamic function pointers in the example I am giving. I will call it unified\_call\_cast for now until we figure out what it should be called. I am not sure if this adds limits to calling conventions or some other unintended consequence but I hope other people can see the problems I might be making here. I feel like I am making an API to convert C++ to C… could/should lambdas work be cast able to a C style function pointer? My first look at this I don’t see why unified\_call\_cast should be as much as a problem as true unified\_call everywhere.

class entity;

typedef void(\*as\_normfun)(entity \*\_this, float y);

typedef void (entity::\*memfun)(float y) const;

memfun mf = &entity::Update;

entity e;

My current code.

as\_normfun snf = (as\_normfun)(&entity\_lerp\_fast\_impl::Update);

as\_normfun dnf = (as\_normfun)(e.\*mf);

assert(snf == dnf);

could become

as\_normfun snf = unified\_call\_cast<as\_normfun>(&entity\_lerp\_fast\_impl::Update));

as\_normfun dnf = unified\_call\_cast<as\_normfun>(e.\*mf);

assert(snf == dnf);

d) It may not work at all on old vtable implementations where vtables entries used to a be a pair or triples of a pointer and offset, or something else; before invention of thunks. Good old cfront used to do this, as did several other contemporary implementations. I think all modern compilers use thunks, but I wonder whether the impossibility of supporting non-thunk-implementations will be a deal-killer.

Until I understand the thunk problem better I don’t know how to reply to this. I need some tests and would like some idea on how to create these tests.