How are we doing? Please help us improve Stack Overflow.
Take our short survey

Performance penalty for working with interfaces in C++?

Asked 16 years, 3 months ago Modified 1 month ago Viewed 19k times

52

Is there a runtime performance penalty when using interfaces (abstract base classes) in C++?

c++ performance abstract-class virtual-functions

Share
Improve this question
Follow

edited Jan 16, 2009 at 9:03
Suma
34.3k ● 18 ● 129 ● 199

asked Sep 22, 2008 at 8:41
andreas buykx
12.9k ● 11 ● 64 ● 76

The answers to this question are also related.
– Richard Corden Sep 22, 2008 at 9:28

17 Answers

Sorted by: Highest score (default)

Short Answer: No.

51

Long Answer: It is not the base class or the number of ancestors a class has in its hierarchy that affects it speed. The only thing is the cost of a method call.

A non virtual method call has a cost (but can be inlined)
A virtual method call has a slightly higher cost as you need to look up the method to call before you call it (but this is a simple table look up **not** a search). Since all methods on an interface are virtual by definition there is this cost.

Unless you are writing some hyper speed sensitive application this should not be a problem. The extra clarity that you will recieve from using an interface usually makes up for any perceived speed decrease.

Share Improve this answer Follow

edited Oct 25, 2008 at 20:20

community wiki
2 revs
Loki Astari

2 It is worth nothing that a virtual method call on an object with multiple base classes has a slightly higher cost again than a virtual method call on an object with a single inheritance hierarchy. – Greg Hewgill Sep 22, 2008 at 8:47

Are you sure? Do you have a source for this comment that I can checkout? – Loki Astari Sep 22, 2008 at 8:52

1 The cost is in the conversion. If you have 'D*' and you want to convert to 'B2*', then, where the layout of the class is [B1, B2] the compiler needs to return 'D* + offset to B2'. I disagree

that this is worth noting however - it will be insignificant.
– Richard Corden Sep 22, 2008 at 9:25

1 May even be zero, depending on the addressing modes offered by your CPU. Accessing register+offset is not unusual. – Steve Jessop Sep 30, 2008 at 16:20

3 @Martin: Actually a virtual method will be inlined in exactly the situations where a non-virtual method would be -- namely, when the dynamic type of the object can be determined at compile time and the method is inlineable (not too large + declared either in the class definition or with "inline").
– j_random_hacker Apr 2, 2009 at 5:21

28

Functions called using virtual dispatch are not inlined

There is one kind of penalty for virtual functions which is easy to forget about: virtual calls are not inlined in a (common) situation where the type of the object is not know compile time. If your function is small and suitable for inlining, this penalty may be very significant, as you are not only adding a call overhead, but the compiler is also limited in how it can optimize the calling function (it has to assume the virtual function may have changed some registers or memory locations, it cannot propagate constant values between the caller and the callee).

Virtual call cost depends on platform

As for the call overhead penalty compared to a normal function call, the answer depends on your target platform.

If your are targeting a PC with x86/x64 CPU, the penalty for calling a virtual function is very small, as modern x86/x64 CPU can perform branch prediction on indirect calls. However, if you are targeting a PowerPC or some other RISC platform, the virtual call penalty may be quite significant, because indirect calls are never predicted on some platforms (Cf. PC/Xbox 360 Cross Platform Development Best Practices).

Share Improve this answer Follow

edited Jun 30, 2012 at 23:14

answered Sep 22, 2008 at 9:39

Suma
34.3k ● 18 ● 129 ● 199

8 It is incorrect to say that virtual calls are not inlined. Whenever the compiler can determine the final type of an object at compile time, method calls on that object are candidates for inlining. It is only when calling via pointer-to-base that inlining cannot be performed. – j_random_hacker Jan 16, 2009 at 19:03

1 @j_random_hacker Is is only the case that you usually use it as a pointer to base. – Ghita Jun 30, 2012 at 6:20

2 @Ghita: Even then, an optimising compiler may be able to figure out what the dynamic type of the object must be. My guess would be that most compilers can inline the call to `foo()` in `Base* x = new Derived; x->foo();`.
– j_random_hacker Jun 30, 2012 at 9:23

3 @j_random_hacker yes it could. But in a normal use case you usually pass the pointer to base class around. It's

interesting to note though that the simplest cases are handled by the compiler. – Ghita Jul 2, 2012 at 17:37

1 Even pointer to base could be inlined by LTO compilation when a full system analysis is done and the interface is just used for modularization and testability and there is just one subclass. Unfortuantely i'm not sure if this is done by any compiler at the moment. – Lothar Jan 12, 2019 at 5:07

10

There is a small penalty per virtual function call compared to a regular call. You are unlikely to observe a difference unless you are doing hundreds of thousands of calls per second, and the price is often worth paying for added code clarity anyway.

Share Improve this answer Follow

answered Sep 22, 2008 at 8:43

moonshadow
88.9k ● 7 ● 86 ● 121

5

When you call a virtual function (say through an interface) the program has to do a look up of the function in a table to see which function to call for that object. This gives a small penalty compared to a direct call to the function.

Also, when you use a virtual function the compiler cannot inline the function call. Therefore there could be a penalty to using a virtual function for some small functions. This is generally the biggest performance "hit" you are likely to see. This really only an issue if the function is small and called many times, say from within a loop.

Share Improve this answer

Follow

answered Sep 22, 2008 at 8:47

David Dibben
18.7k ● 6 ● 43 ● 41

Don't let it screw up your design, indeed, but only use virtual functions if you really need them - inlining can cause HUGE performance gain when you think of iterating over a (big) number of elements and calling the method on each one of them. – xtofl Sep 22, 2008 at 8:53

1 It is incorrect to say that virtual calls cannot be inlined. Whenever the compiler can determine the final type of an object at compile time, method calls on that object are candidates for inlining. It is only when calling via pointer-to-base that inlining cannot be performed. – j_random_hacker Jan 16, 2009 at 19:04

4

Another alternative that is applicable in some cases is compile-time polymorphism with templates. It is useful, for example, when you want to make an implementation choice at the beginning of the program, and then use it for the duration of the execution. An example with runtime polymorphism

```
class AbstractAlgo
{
    virtual int func();
};

class Algo1 : public AbstractAlgo
{
    virtual int func();
};

class Algo2 : public AbstractAlgo
{
    virtual int func();
};

void compute(AbstractAlgo* algo)
{
      // Use algo many times, paying virtual function

}

int main()
{
    int which;
     AbstractAlgo* algo;

    // read which from config file
    if (which == 1)
       algo = new Algo1();
    else
       algo = new Algo2();
    compute(algo);
}
```

The same using compile time polymorphism

```
class Algo1
{
    int func();
};

class Algo2
{
    int func();
};


template<class ALGO>  void compute()
{
    ALGO algo;
      // Use algo many times.  No virtual function cos
inlined.
}
```

```
int main()
{
    int which;
    // read which from config file
    if (which == 1)
       compute<Algo1>();
    else
       compute<Algo2>();
}
```

Share Improve this answer

Follow

answered Sep 22, 2008 at 13:02

KeithB
17k ● 3 ● 43 ● 45

Unfortunately, this does not apply to plugin classes and other dynamically loaded types (yes, this *is* possible in C++ :-) – André Caron Oct 18, 2010 at 18:49

This adds a lot to code bloat and i still wait for any study that examines the impact of template code bloat on performance (cache problems, branch predictions ...). – Lothar Jan 12, 2019 at 5:10

3

I don't think that the cost comparison is between virtual function call and a straight function call. If you are thinking about using a abstract base class (interface), then you have a situation where you want to perform one of several actions based of the dynamic type of an object. You have to make that choice somehow. One option is to use virtual functions. Another is a switch on the type of the object, either through RTTI (potentially expensive), or adding a type() method to the base class (potentially increasing memory use of each object). So the cost of the virtual function call should be compared to the cost of the alternative, not to the cost of doing nothing.

Share Improve this answer Follow

answered Sep 22, 2008 at 12:42

KeithB
17k ● 3 ● 43 ● 45

3

Most people note the runtime penalty, and rightly so.

However, in my experience working on large projects, the benefits from clear interfaces and proper encapsulation quickly offset the gain in speed. Modular code can be swapped for an improved implementation, so the net result is a large gain.

Your mileage may vary, and it clearly depend on the application you're developing.

Share Improve this answer Follow

answered Sep 22, 2008 at 17:00

rlerallut
7,829 ● 5 ● 25 ● 21

I'd like to offer the counter point: For embedded systems (others have mentioned video game consoles), the performance hit might be too large and unless there's alot of be gained from maintainability/readability/clarity, etc. they should be avoided. – It'sPete Jul 30, 2013 at 22:07

Note that multiple inheritance bloats the object instance with multiple vtable pointers. With G++ on x86, if your

3

class has a virtual method and no base class, you have one pointer to vtable. If you have one base class with virtual methods, you still have one pointer to vtable. If you have two base classes with virtual methods, you have *two* vtable pointers *on each instance*.

Thus, with multiple inheritance (which is what implementing interfaces in C++ is), you pay base classes times pointer size in the object instance size. The increase in memory footprint may have indirect performance implications.

Share Improve this answer Follow

answered Oct 12, 2008 at 11:42

hsivonen
8,006 ● 1 ● 35 ● 36

2

One thing that should be noted is that virtual function call cost can vary from one platform to another. On consoles they may be more noticeable, as usually vtable call means a cache miss and can screw branch prediction.

Share Improve this answer Follow

answered Sep 22, 2008 at 9:38

yrp
4,575 ● 2 ● 26 ● 10

---

2 I've heard this stated often, but never actually validated. The cost effective cost on global performance application performance, in consoles or not, actually depends on the frequency at which you call the method. Most of my friends in the gaming industry are not allowed to use virtual functions *at all* on behalf of your argument. They end up with messed up

code to circumvent this limitation even for higher-level function called once per render cycle. – André Caron Oct 18, 2010 at 18:46

0

Using abstract base classes in C++ generally mandates the use of a virtual function table, all your interface calls are going to be looked up through that table. The cost is tiny compared to a raw function call, so be sure that you need to be going faster than that before worrying about it.

Share Improve this answer Follow

answered Sep 22, 2008 at 8:45

Simon Steele
11.6k ● 4 ● 47 ● 67

0

The only main difference I know of is that, since you're not using a concrete class, inlining is (much?) harder to do.

Share Improve this answer Follow

answered Sep 22, 2008 at 8:45

C. K. Young
223k ● 47 ● 390 ● 443

0

The only thing I can think of is that virtual methods are a little bit slower to call than non-virtual methods, because the call has to go through the virtual method table.

However, this is a bad reason to screw up your design. If you need more performance, use a faster server.

Share Improve this answer

Follow

answered Sep 22, 2008 at 8:45

jan.vdbergh

2,119 ● 2 ● 20 ● 29

0

As for any class that contains a virtual function, a vtable is used. Obviously, invoking a method through a dispatching mechanism like a vtable is slower than a direct call, but in most cases you can live with that.

Share Improve this answer

Follow

answered Sep 22, 2008 at 8:48

radu_c

147 ● 1 ● 4

0

Yes, but nothing noteworthy to my knowledge. The performance hit is because of 'indirection' you have in each method call.

However, it really depends on the compiler you're using since some compilers are not able to inline the method calls within the classes inheriting from the abstract base class.

If you want to be sure you should run your own tests.

Share Improve this answer

Follow

answered Sep 22, 2008 at 8:53

mortenbpost

1,707 ● 16 ● 22

0

Yes, there is a penalty. Something which may improve performance on your platform is to use a non-abstract class with no virtual functions. Then use a member function pointer to your non-virtual function.

Share Improve this answer Follow

answered Sep 22, 2008 at 14:24

Juan

0

I know it's an uncommon viewpoint, but even mentioning this issue makes me suspect you're putting way too much thought into the class structure. I've seen many systems that had way too many "levels of abstraction", and that alone made them prone to severe performance problems, not due the cost of method calls, but due to the tendency to make unnecessary calls. If this happens over multiple levels, it's a killer. take a look

Share Improve this answer Follow

answered Nov 4, 2008 at 22:34

Mike Dunlavey
40.6k ● 15 ● 94 ● 138

0

I have an application that runs a virtual machine to compute derivatives of functions. A function is made up of many operations. (For example, MulvvOp is the operator that adds two variables.) In order to simplify the loop over the operators I tried using a base class and a virtual function for the operators, instead of a separate case for each operator.

The performance hit was usually between 4% and 15%.

1. The names that end with _one corresponds to separate calls operator in different switch cases. These the computation for each operator are declared inline.
2. The names that end with _two corresponds to one case in the switch (for 46 of the operators) and virtual function calls.
3. The rate is the number of times per second that the corresponding derivatives could be calculated.
4. The components of the vector correspond to the number of operations in the function being tested (from smaller to larger).

Results:

```
det_lu_rate_one = [ 855294, 5909, 795.27, 244.47, 101.
det_lu_rate_two = [ 847786, 5199, 697.96, 212.82, 87.8
det_minor_rate_one = [ 570547, 313913, 102593, 21601,
det_minor_rate_two = [ 558110, 299863, 98528, 20136, 3
mat_mul_rate_one = [ 981139, 3691, 507.86, 133.43, 59.
mat_mul_rate_two = [ 907647, 3231, 446.07, 119.22, 53.
ode_rate_one = [ 177964, 1077, 306.69, 142.72, 82.34 ]
ode_rate_two = [ 171636, 757.62, 207.71, 95.17, 54.34
poly_rate_one = [ 1001246, 534508, 375909, 291816, 239
poly_rate_two = [ 976578, 490359, 334493, 258417, 2083
sparse_hessian_rate_one = [ 392.83, 32.47, 6.8397, 2.0
sparse_hessian_rate_two = [ 380.60, 34.43, 7.3060, 2.1
sparse_jacobian_rate_one = [ 1256, 277.47, 91.90, 29.8
sparse_jacobian_rate_two = [ 1008, 228.71, 85.56, 28.6
```

For example, dividing the last entry in the first and second row below we get a hit of about 15% :

```
det_lu_rate_one(4) / det_lu_rate_two(4) = 1.1487
```

Share Improve this answer

Follow

answered Oct 26 at 13:34

Brad Bell

49 ● 3