How do multiple levels of inheritance affect virtual function call overhead in C++?

Ask Question

3

1

I am considering using a class hierarchy with more than a single level of inheritance, where virtual member functions form a "chain", for example:

```
struct Base
{ virtual void foo(); };

struct D1 : Base
{ virtual void foo(); };

struct D2 : D1
{ void foo(); };
```

However I would like to know before using similar code in practice if this would cause additional dynamic dispatch overhead when calling trough a base class pointer. Consider the following example:

```
D2 instance;
D1* d1ptr = &instance;
Base* baseptr = &instance;

//normal function call to "D2::foo()", no dynamic dispatch.
instance.foo();
//virtual function call to "D2::foo()", one dynamic dispatch.
d1ptr.foo();
//virtual function call to "D2::foo()", does this incur additio
//..overhead compared to "d1ptr.foo()"?
baseptr.foo();
```

c++ functions polymorphism

share improve this question

asked Sep 19 '14 at 23:56

jms

138 1 5

add a comment

1 Answer

active oldest votes

7

In your example, the depth of the inheritance tree does not affect performance. The reason is simple, every instance has its pointer to some vtable (containing function pointers), and a virtual function call goes just thru that vtable.

In other words, your code works like the following C code:

```
struct vtable {
   void (*fooptr) (struct D1*);
};

struct D1 {
   const struct vtable*vptr;
};

struct D2 {
   const struct vtable*vptr;
};
```

And e.g. `baseptr->foo()` is "transformed" at compilation time to `baseptr->vptr.fooptr(baseptr)` (and of course `d1ptr->foo()` is "transformed" to `d1ptr->vptr.fooptr(d1ptr)` etc...)

So the execution cost is the same: get the vtable, and call *indirectly* the function pointer inside. The cost remains the same even if you had a `struct D4` subclass of `struct D3` subclass of `struct D2` subclass of `struct D1` . You could have a `struct D99` with 99 inheritances and the execution cost would remain the same.

On some processors, any *indirect call* may be slower than a *direct* call, e.g. because of branch predictor cost.

The data of the vtable themselves is built at compilation time (as constant static data).

Things become a bit more complex with virtual inheritance and/or multiple inheritance

BTW, some C object frameworks (e.g. GObject from Gtk, or several data structures inside the Linux kernel) also provide their vtable (or class data). Pointing to a data structure containing function pointers is quite common (even in C). Read also about closures and you'll see a relation with "vtables" (both closures and objects are mixing code with data).

Some C++ compilers (e.g. GCC) may provide devirtualization as an optimization technique.

share improve this answer

edited Sep 20 '14 at 6:19

answered Sep 20 '14 at 5:47

Basile Starynkevitch

27.9k 5 62 102

C++ 11 provides keyword final, does it not, which would make devirtualization of leaf functions straightforward. I have seen gcc devirtualize without final, playing around on the godbolt page, but it's

almost hilariously fragile and cannot be relied on. – Nir Friedman Sep 21 '14 at 16:42

1 @underscore_d `B` inherits from `A`, they both define a virtual function `bar`. A function `foo`, part of a shared library, receives a parameter `B& b`, and calls `b.bar();`. This can be devirtualized only if either `B` or `bar` are marked `final`. But if they are, then devirtualization is trivial and most modern compilers will perform it. – Nir Friedman Jul 25 '16 at 15:18

1 @underscore_d Even in self-contained programs, this kind of optimization can only be done by the linker, not the compiler, and scales very poorly with program size. I'm guessing there are also other deep dark corners. I think the bottom line is that you can rarely expect to see this when inlining does not occur. As I said, devirtualization w/o final is extremely fragile in my experience. Have you actually seen consistent devirtualization without final, without inlining? It seems like your discussion is more theoretical. – Nir Friedman Jul 25 '16 at 18:46

1 @NirFriedman Yup, it's not something I've profiled yet, so I was just looking for rationales and evidence - or at least anecdotes - supporting it. So I guess I'll take your word for it! If good practices like using `final` lead to any concrete improvements in generated code, then that's a double win. – underscore_d Jul 25 '16 at 19:47

1 @NirFriedman Here's a good little demo of the ability of `final` to facilitate optimisations as you said - msinilo.pl/blog/?p=1329 - at least for GCC and Clang, dunno about The Other Compiler, but it didn't take advantage of this at the time that was written, *quelle surprise* – underscore_d Jul 30 '16 at 17:38