Zero-cost exceptions aren’t actually zero cost

devblogs.microsoft.com/oldnewthing/20220228-00

February 28, 2022

Raymond Chen

There are two common models for exception handling in C++. One is by updating some program state whenever there is a change to the list of things that need to be done when an exception occurs, say, because a new exception handler is in scope or has exited scope, or to add or remove a destructor from the list of things to execute during unwinding. Another model is to use metadata to describe what to do if an exception occurs. There is no explicit management of the state changes at runtime; instead, the exception machinery infers the state by looking at the program counter and consulting the metadata.

Metadata-based exception handling is often misleadingly called *zero-cost exceptions*, which makes it sound like exceptions cost nothing. In fact, it’s the complete opposite: Metadata-based exception handling should really be called *super-expensive exceptions*.

The point of metadata-based exception handling is that there is no code in the mainline (non-exceptional) code path for exception support. The hope is that exceptions are rare, so you end up with a net win:

Mode	**Runtime-managed**	**Metadata-based**
Mainline code	Update state at runtime	
Exception occurs	Consult the state to find the correct handler	Take the program counter, find the metadata that applies to it, consult the metadata to find the correct handler

Notice that using metadata-based so-called “zero-cost” exceptions actually results in a significantly *higher* cost for throwing an exception, because the exception-throwing machinery has to go find the metadata so it can look up which handler to run. This metadata is typically stored in a format optimized for size, not speed, so extra work has to happen at exception-throwing time to decode the data in order to find the correct handler.

The name “zero-cost exceptions” refers to the empty box in the upper right corner. There is no code generated to maintain state just in case an exception occurs.

But even though the box is empty, that doesn't mean that things are still the same as if there were no exceptions.

The presence of exceptions means that the code generation is subject to constraints that don't show up explicitly in the code generation: Before performing any operation that could potentially throw an exception, the compiler must spill any object state back into memory if the object is observable from an exception handler. (Any object with a destructor is observable, since the exception handler may have to run the destructor.)

Similarly, potentially-throwing operations limit the compiler's ability to reorder or eliminate loads from or stores to observable objects because the exception removes the guarantee of mainline execution.

These costs are not visible to the naked eye. They take the form of lost optimization opportunities.

Zero-cost exceptions are great (despite the blatant misnomer), but be aware that the cost is not actually zero.

Raymond Chen

Follow