Throw2goto: A step towards zero-overhead dynamic exceptions

“There ain't no such thing as the fastest code.”

* *Michael Abrash*

**Date:** April 28, 2020

**Document Number:** idk

**Project:** Clang/LLVM

**Audience:** LLVM-dev

**Reply-To:** Jeff Linahan [jeff.linahan@gmail.com](mailto:jeff.linahan@gmail.com)

**Introduction**

We propose an exception handling optimization that achieves zero overhead in both space and time compared to ordinary control flow for some common try-catch block patterns. The technique brings exceptions more in line with the design goals of C++, reducing the need for alternate error handling mechanisms.

**Motivation**

After decades of existing as a supported language feature, exceptions are still not universally used as an error handling mechanism in C++ despite being carefully designed around RAII. The purpose was to allow a way to signal an error that cannot be handled locally. One commonly cited reason for low usage is that exceptions violate the zero-overhead principle.  Recall that it states:

1. You don’t pay for what you don’t use
2. What you do use is just as efficient as what you could [reasonably][[1]](#endnote-1) write by hand

This applied to both time and space. Note that zero overhead does not mean zero cost. Rather, if a certain C++ language feature is used, the code must be just as efficient as what a programmer could produce coding without using that feature.

Early exception handling implementations did not meet this principle; much bookkeeping is done to make sure the correct destructors are called in the event an exception is thrown.

Is it enough that a feature simply be possible to implement in a zero-overhead manner? Sutter has proposed static exceptions as a possible way to to implement exceptions in the spirit.

The key difficulty in producing a zero-overhead exceptions implementation is that C++’s exceptions are actually extremely powerful. Objects of any type (including built ins) may be thrown. They may even be of a polymorphic type not known until runtime. It may not be possible to tell what the call graph looks like at compile time as control flow may pass through virtual function and function pointer boundaries. In C++, the exception handling mechanism must find the correct catch block (or landingpad) needs to be found, even for non-polymorphic (i.e. no virtual functions)

Goldthwaite has done a comprehensive examination of the kinds of runtime costs of exception handling and identified three sources of overhead in both data and code: (1) try and catch blocks, (2) regular functions, and (3) throw-expressions. He compares the “code” approach with the “table” approach. In the best case, code equipped with exception handling can outperform even comparable C code because many if-checks can be omitted. In the worst case scenario in the table approach, a throw-expression may cause a disk access to load state tables from virtual memory.

For this reason, the author believes that (within reason) the lack of a sufficiently smart compiler should not be a reason for so much of the community to turn off dynamic exceptions when there is still room for improvement.

Note the calls to \_\_cxa\_allocate\_exception and \_\_cxa\_throw, hereafter we refer to this as the *dynamic dispatch mechanism.* Due to the fact that C++ allows multiple active exception objects of arbitrary types, it requires some form of RTTI to match handles at runtime [Sutter]. This kind of type matching may be completely orthogonal to the kind used in type\_info and dynamic\_cast, but still imposes a space overhead in the form of jump tables, additional stack space, and thread\_local storage.

Part (1) many developers use -fno-exceptions to prevent the compiler from generating RTTI data and part (2) the generated assembly code is often larger and slower than using other control flow mechanism.

The purpose of exception handling is to signal an error that cannot be handled locally, thus decoupling the code that detects it from the code that (attempts to) recover from it. Where

However, much exception handling code is similar to the C-style of if-checks and a cleanup clause at the end of a function. Modern compilers are able to implement exceptions with no runtime overhead when everything is going fine at the expense of a severe slowdown during the stack unwinding process when a throw occurs. For this reason, many developers continue the C-style error checking of if’s and goto’s for errors that can be handled locally, often going so far as to disable exceptions entirely. This unfortunately means that if’s must be used everywhere, thus missing out on zero-overhead in the happy case.

Much C-style error checking code has a cleanup label at the end of a function and several goto statements jumping into it. With this throw2goto optimization, try-catch blocks containing throws that actually catch the exception without letting it propagate up (down?) the stack do not invoke the dynamic dispatch optimization.

**Low-hanging Fruit**

Is it such a pound of cure to improve current exception implementations? Is it really so against the spirit of C++ to expect certain optimizations to occur? C++ has mandated optimizations to happen before, for example return value optimization (RVO) in certain cases starting in C++17. Even in Scheme, a language so dynamic there is little difference between development time and runtime, mandates tail-call optimization.

While progress has certainly been made for when exceptions are not being thrown, there is still room for improvement when they are. In practice there are many throw’s where we can see the catch. Consider this program:

int main() try {throw 42;} catch (int e){}

With -O3, clang generates exception handling code:

main:                                   # @main

        push    rax

        mov     edi, 4

        call    \_\_cxa\_allocate\_exception

        mov     dword ptr [rax], 42

        mov     esi, offset typeinfo for int

        mov     rdi, rax

        xor     edx, edx

        call    \_\_cxa\_throw

        mov     rdi, rax

        call    \_\_cxa\_begin\_catch

        call    \_\_cxa\_end\_catch

        xor     eax, eax

        pop     rcx

        ret

With proper optimization, this should get changed to:

main:                                   # @main

        xor     eax, eax

        Ret

If we std::cout the int to prevent the optimizer from eliding it, we have:

#include <iostream>

int main(){

    try{

        throw 42;

    } catch(int e){

        std::cout << e;

    }

}

And again with -O3, we can see the stack unwinding call:

main:                                   # @main

        push    rbx

        mov     edi, 4

        call    \_\_cxa\_allocate\_exception

        mov     dword ptr [rax], 42

        mov     esi, offset typeinfo for int

        mov     rdi, rax

        xor     edx, edx

        call    \_\_cxa\_throw

        mov     rbx, rax

        cmp     edx, 1

        jne     .LBB0\_5

        mov     rdi, rbx

        call    \_\_cxa\_begin\_catch

        mov     esi, dword ptr [rax]

        mov     edi, offset std::\_\_1::cout

        call    std::\_\_1::basic\_ostream<char, std::\_\_1::char\_traits<char> >::operator<<(int)

        call    \_\_cxa\_end\_catch

        xor     eax, eax

        pop     rbx

        ret

        mov     rbx, rax

        call    \_\_cxa\_end\_catch

.LBB0\_5:

        mov     rdi, rbx

        call    \_Unwind\_Resume

Which is still a far cry from:

main:                                   # @main

        push    rax

        mov     edi, offset std::\_\_1::cout

        mov     esi, 42

        call    std::\_\_1::basic\_ostream<char, std::\_\_1::char\_traits<char> >::operator<<(int)

        xor     eax, eax

        pop     rcx

        ret

The case of throwing int is not as niche as it might first appear, as throwing errno when mixing C and C++ is a common practice in embedded systems. Indeed, when throwing any built-in type we can simply gensym\* an identifier to hold the value and substitute it for the caught exception in the catch block. For the case of throwing nothing, if an exception is currently not being thrown (like in a function declared noexcept) the throw can simply be replaced with std::terminate(). If an exception is being thrown the \_\_cxa\_rethrow call remains.

We should note the output of int main(){throw;}

main:                                   # @main

        push    rax

        call    \_\_cxa\_rethrow

This \_\_cxa\_rethrow call can be changed to std::terminate() or the xor and return like above.

**Easy cases**

If there are no objects with non-trivial destructors created in a try block and the type can be deduced at compile time, any throw’s can be replaced with goto’s into the catch block. The exception object can be created with gensym’d symbol.

**Limitations**

Similar to speculative devirtualization, doing this type of optimization across function and translation unit (TU) boundaries is very hard, often requires whole program information paired with link time optimization in general.

Examining one function containing a try-catch block in isolation, it is not possible to determine what exceptions may be thrown without exception specifications (which were deprecated in C++11) because the calls in the try block may have arbitrary logic and throw any type.

Examining one TU in isolation, throw2goto cannot (in general) be applied to functions designed to throw and catch with each other, whether by means of inlining the throwing function or changing the throw to a fancy call-destructors-popstack-and-goto because polymorphic types can be thrown and the landingpad address cannot in general be known at compile time. Also, because another translation unit may call the throwing function, so the dynamic dispatch mechanic needs to be there. Only static or private non-friend functions are immune from this.

Much like template patterns are used to generate multiple copies of object code and inlining generates extra copies of the function body, small throwing functions that are inlined can make such throw2goto optimizations possible, but the complexity lies striking the balance between compiler time and code size for different optimization levels.

**Acknowledge**

Jeff thanks Emil Dotcheviski for helpful discussions on how exception handling is implemented, feedback on the ideas presented here, and proofreading.

**Conclusion**

Although there are many cases that cannot be optimized to goto, many of them either do not occur in practice or would incur similar overhead in vanilla C.

**References**

Sutter, Herb.Zero-overhead deterministic exceptions: Throwing values. P0709 R0. 2018-05-02. <http://open-std.org/JTC1/SC22/WG21/docs/papers/2018/p0709r0.pdf>

Goldthwaite, Lois. *Technical Report on C++ Performance*. WG21/N1666 J16/04-0106. 2004-7-15. <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2004/n1666.pdf>

<http://www.stroustrup.com/abstraction-and-machine.pdf>

<http://www.stroustrup.com/bs_faq.html#Hello-world>

Koenig, Andrew. <http://www.stroustrup.com/except89.pdf>

Inline assembly generated by Compiler Explorer: <https://godbolt.org/>

1. There are multiple interpretations of the zero-overhead principle, sometimes quoted with and without the word “reasonable.” For this paper we can consider “reasonable” to be that which could be produced by a skilled C programmer with knowledge of the target architecture. Rice’s theorem and the full employment theorem offer some perspective on why it is not possible in general for a compiler to tell whether it has generated truly optimal code. [↑](#endnote-ref-1)