Throw2goto: An exception handling optimization

Jeff Linahan

Abstract

We discuss an exception handling optimization that achieves zero overhead in both space and time compared to ordinary C-style error handling control flow when the compiler can see which catch block a given throw expression will land in. The technique brings exceptions more in line with the design goals of C++, reducing the need for alternate error handling mechanisms.

Throw2goto: an exception handling optimization

Jeff Linahan

Introduction

We discuss an exception handling optimization that achieves zero overhead in both space and time compared to ordinary C-style error handling control flow when the compiler can see which catch block a given throw expression will land in. 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 accepted as C++'s de facto error handling mechanism, despite being carefully designed around RAII. Perhaps the most commonly cited reason for low adoption is that exceptions violate the **zero-overhead principle**. Early exception handling implementations fell far short of achieving this as (among other things) much bookkeeping was done to make sure the correct destructors of objects with automatic storage duration were called just in case an exception is thrown; this would happen even during the "happy path" when no exceptions are occurring. This led many developers to disable exceptions entirely and led to the proliferation of many different styles of error handling in C++, even putting pressure on the committee to support different dialects of the language (for example with `std::error_code` in `std::filesystem`).

While progress has certainly been made on improving the performance of exceptions, this has often been in the form of trading off one performance problem for another. Earlier, Goldthwaite has done a comprehensive examination of the kinds of runtime costs of exceptions and identified three sources of overhead, each in both data and code: (1) bookkeeping associated with each try and catch block (to remember to call the correct destructors, for example) (2) missed opportunities for optimization in regular functions that exceptions prevent us from applying, and (3) overhead associated with the actual throwing, for example dynamic allocation of the exception object. He compares the "code" (or stack-based) approach with the "table" approach. In the table-based approach, the best case scenario is that code equipped with exception handling can outperform even comparable C code because the "error propagation clutter" of many "if" checks on error codes can be omitted from the deeply nested (or telescoping recursive) function calls. [Cline] Note that this is a negative overhead abstraction, similar to coroutines for single-pass Cobol compilation [Nishanov] or matrix multiplication in Fortran [Fridman]. In the worst case scenario however, a throw may cause a disk access to load state tables from cache-cold virtual memory.

For those like myself who would like more exception handling in the wild, do we simply continue trying to convince developers merely not overuse exceptions instead of

completely abandoning them when so much of the community has valid complaints? Are we stuck with waiting for contracts to be merged into the standard, `std::logic_error` to be deprecated, followed by decades for the industry to refactor away the overuse until we are left with exceptions thrown only in truly "exceptional" situations? Do we then sum the negative overhead of the best case with the disk access of the worst, hoping for a nonpositive result so we can claim the zero overhead principle is preserved?

Dynamic Difficulties

Part of the difficulty in producing a zero-overhead exceptions implementation comes from the fact that C++'s exceptions are actually extremely powerful, and inherently dynamic. Objects of any type (including built-ins and polymorphic types) may be thrown, they can propagate through virtual function and function pointer boundaries, and the runtime must be able to determine whether the exception object's type matches a particular catch clause. Finding the address of the activation handler (or landingpad) at compile time is (in general) not possible and requires whole program information in many cases. It may not be possible to tell what the call graph looks like at compile time if control flow passes through these, and a base class catch block must match a thrown derived object, even if it is non-polymorphic.

Inherent runtime polymorphic issues aside, it is not enough that a feature simply be possible to implement in a zero-overhead manner; the standard has seen dizzyingly complex features before (such as exported templates) that are impractical to implement. Static exceptions have been proposed as a new mechanism that follows the spirit behind the zero overhead principle in a more straightforward way by having the return channel do double duty as the exception object's memory. [Sutter] But there are clearly many programs that use dynamic exceptions that need to keep working. Is it such a pound of cure to improve the current mechanism? Is it really so against the spirit of C++ to use optimizations to achieve zero-overhead, or at least try to? C++ has before mandated certain optimizations to happen before, for example return value optimization.

Local Landingpads

The design purpose of exceptions 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. This does not mean all exceptions must be thrown from a different stack frame in which they are caught: try blocks that contain throw statements landing in the immediately following catch block are also very common. For most of these throw statements, the thrown object's type is known at compile time, yet modern compilers still generate code for dynamic allocation and dynamic dispatch, thus the overhead is higher than what a C programmer would write with a `goto errorhandler` statement to jump to the end of the function. To see a dramatic example of this cost, consider:

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

With -O3, clang generates code to dynamically allocate the exception object, look up typeinfo, throw it, and does bookkeeping as the catch block begins and ends (even when there are no instructions in the catch block):

```
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 be simply:

```
main:                                   # @main
        xor     eax, eax
        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. Of course, usually we have code before (and after) the throw:

```
void update_warp_factor(float speed)
{
    try {
        float w = calc_warp_factor(speed);
        if (w >= 10.0) throw WarpCoreException("bad speed");
        if (w <= 1) switch_to_impulse_engine();
    } catch (WarpCoreException const& e){
        cerr << e.text << '\n';
    }
}
```

In this case, a sufficiently smart compiler can see that the thrown `WarpCoreException` always lands in the following try block, so there is no need for a dynamic memory allocation, and the catch clause is replaced with a `landingpad` label. Here is how we convert the throw statement: Since the WarpCoreException's lifetime is limited to this function, we can `alloca` it on the stack, which simply increments the stack pointer by `sizeof(WarpCoreException)` and gives us an area of uninitialized memory in which we can construct the `WarpCoreException`. (Care must be taken to make sure the exception object is not too big to prevent a stack overflow). Then we `goto`

`landingpad`. After the throw2goto optimization is applied, we are left with an AST describing something like this:

```
void update_warp(float speed)
{
        WarpCoreException* __ex = nullptr;

        float w = calc_warp_factor(speed);

        if (w >= 10.0) {
              __ex = static_cast<WarpCoreException*>(alloca(sizeof(WarpCoreException)));
              new (__ex) WarpCoreException("bad speed");
              goto landingpad;
       }

       if (w <= 1) switch_to_impulse_engine();
       return;

landingpad:
       cerr << __ex->text << '\n';
       __ex->~WarpCoreException();
}
```

Here, __ex denotes an arbitrary C++ identifier that is guaranteed to not cause an ODR violation. For now, the optimization can simply be disabled in the presence of language idioms that make the transformation difficult to apply correctly, such as `throw;` for the Lippincott pattern, or `std::current_exception` and `std::uncaught_exceptions` rather than adding one to count the “optimized out” exception with the as-if rule.

More Ambitious Ideas

What if there are multiple throw statements in the try block? So long as none of the throws are polymorphic, we create an additional block that performs the alloca, placement new, and goto for each throw, and an additional unique landingpad label for every type.

What if some of the helper functions (with internal linkage) like `calc_warp_factor()` or `switch_to_impulse_engines()` throw an exception that lands in `update_warp()`’s catch block? Perhaps we can still optimize this too: one idea is to use multiple return addresses. In addition to the normal return address, the address of each landingpad could be pushed onto the stack so the callee functions could restore the program counter to different points depending on which throw statement was encountered. If we are several function calls deep and there are no objects with nontrivial destructors allocated in between, we need not even worry about bit-blitting [Sutter, 2019] the exception object up each of the n frames during stack unwinding, but can catch in constant time.

What if a function `g()` is called from within the catch block and `g()` also throws? This is known to require whole-program information. [cfe-dev 042051] Many of more complex

cases are like this, but as link time optimization becomes more popular, it is hoped that eventually many of them can be applied.

Conclusion

Exception handling is still hotly debated despite being standard C++ and used throughout the STL, and many alternative error handling mechanisms exist. Perhaps the biggest complaint is the performance, and while much progress has certainly been made on the happy path, the low hanging fruit of optimizing throwing into a local landingpad has not yet been implemented. There is higher hanging fruit as well, but if we begin optimizing some common exception phraseologies to be truly zero overhead compared to C-style return codes, perhaps we will we less community fragmentation and more developers able to use the language's foremost error handling facility.

Acknowledgement

Many thanks to Emil Dotchevski for helpful discussions on how exception handling is implemented, feedback on the ideas presented here, and proofreading. Inline assembly generated by Compiler Explorer: https://godbolt.org/

References

Cline, Marshall. *C++ Super-FAQ: Exceptions and Error Handling.* https://isocpp.org/wiki/faq/exceptions#exceptions-avoid-spreading-out-error-logic

Fidman, Lex & Stroustrup, Bjarne. *C++ | Artificial Intelligence Podcast*. https://www.youtube.com/watch?v=uTxRF5ag27A

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

Luehring, Dennis. *Why does clang(llvm) not optimize static throw/catch constructs?* http://lists.llvm.org/pipermail/cfe-dev/2015-March/042051.html

Koenig, Andrew. *Exception Handling for C++*. http://www.stroustrup.com/except89.pdf

Nishanov, Gor. *C++ Coroutines – a negative overhead abstraction*. https://www.youtube.com/watch?v=_fu0gx-xseY

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

Sutter, Herb. *De-fragmenting C++: Making Exceptions and RTTI More Affordable and Usable.* CppCon 2019. https://www.youtube.com/watch?v=ARYP83yNAWk