C++ Exception Optimizations. An experiment.

Document Number: **P 1676 R0** Date: 2019-06-04
Reply-to: Gor Nishanov (gorn@microsoft.com)
Audience: EWG

Abstract

This paper reports a positive experience in developing optimizations that look for several C++ exceptions throw and catch patterns and replace exception machinery with regular control flow.

While the original motivation for exploring optimization of these patterns was that they frequently occur in coroutines, these optimizations may be beneficial for other application categories and more optimizations of this kind can be developed if this approach proves profitable.

Contents

1 Introduction

There are few C++ exceptions related optimization in modern compilers. One common optimization is to discover which functions cannot throw at all and propagate that information through a call graph, eliminate exception handling in the functions that will not ever see an exception. This paper looks at another set of cases where functions do throw and optimizations transform the code either by completely replacing EH machinery with regular control flow or replace a costlier mechanism such as catch(…) and rethrow with less expensive unwind/cleanup logic.

There are several language features, specifically “throw;”, “std::current_exception()” and “std::uncaught_exceptions()”, that interact with the described optimizations. We will start with a simple case where optimizer can observe that no calls to those facilities is made on the optimized code path and cover the details of handling the tricky cases in the “Complications” section.

2 Catch and rethrow optimization

Functions that do not have a try-catch and rely on destructors to do the cleanup when the exception is thrown have less overhead during exception propagation that functions that do have a catch and rethrow. This makes it profitable to perform this transformation:

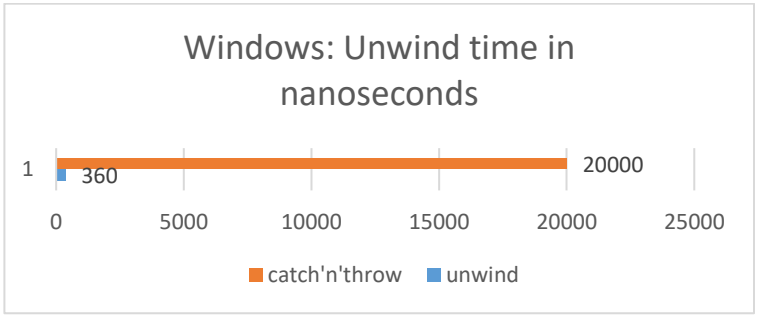
Before

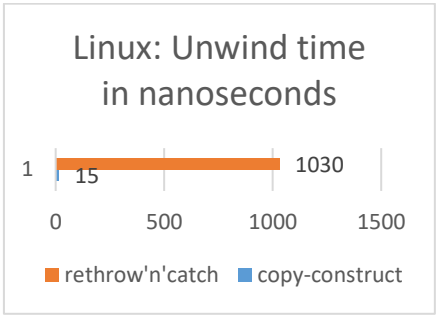
```
{
  SomeType someVar;
  try {
    may_throw();
  } catch (...) {
    payload();
    throw;
  }
}
```

After[1]

```
{
  SomeType someVar;
  auto _= std::make_scope_fail([]{
    payload();
  });
  may_throw();
}
```

This results in 55x speedup on Windows amd64 and 70x speed up on Linux amd64 for synthetic benchmark that runs an inexpensive cleanup (an increment of a variable). Most of the measured cost is the exception handling.





This pattern occurs after inlining in synchronous generator coroutines that allow an exception to propagate to the user of the generator whenever he/she pulls the next value. It is profitable to run this optimization if a function is a coroutine.

[1] Here we expressed pushing the payload on a cleanup path as if we used a scope guard facility from [P0052]. Actual optimization obviously does not use a scope guard and only manipulates the intermediate representation so that the effect of unwind from that point would be to call the payload().

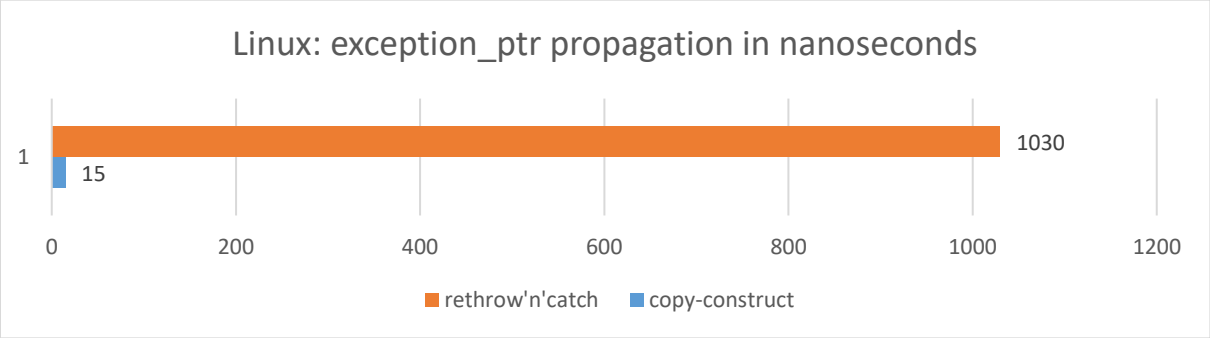
3 Propagate exception pointer optimization

Another pattern that occurs after inlining in the async code dealing with futures, promises and/or asynchronous coroutines is rethrowing an exception from exception_ptr, catching it with catch(…) and storing the same exception elsewhere. This makes it profitable to perform this transformation:

Before	After

`try {`
`  SomeType someVar;`
`  std::rethrow_exception(src);`
`} catch (...) {`
`  dst = std::current_exception();`
`}`
	`{`
	`  SomeType someVar;`[2]
	`}`
	`dst = src;`[3]

This results in 70x speedup on Linux amd64 for synthetic benchmark that runs an inexpensive cleanup (an increment of a variable). Most of the measured cost is the exception handling.



More general form of this optimization is as follows:

Before	After

`try {`
`  SomeType someVar;`
`  may_throw();`
`  std::rethrow_exception(src);`
`} catch (...) {`
`  dst = std::current_exception();`
`}`
…
	`  try {`
	`  SomeType someVar;`
	`  may_throw();`
	`  goto save_eptr;`
	`  } catch (...) {`
	`  dst = std::current_exception();`
	`  }`
	`  goto save_bypass;`
	
	`save_eptr: dst = src;`
	`save_bypass:`
	…

[2] See “Complications” section that explains how to deal with destructors calling tricky functions that may observe this transformation.

[3] The actual code pattern observed is not an assignment, but a copy construction of an exception in the unitialized storage for destination, such as `new (&dst_storage) exception_ptr(current_exception());` . We showed dst = src for exposition simplicity.

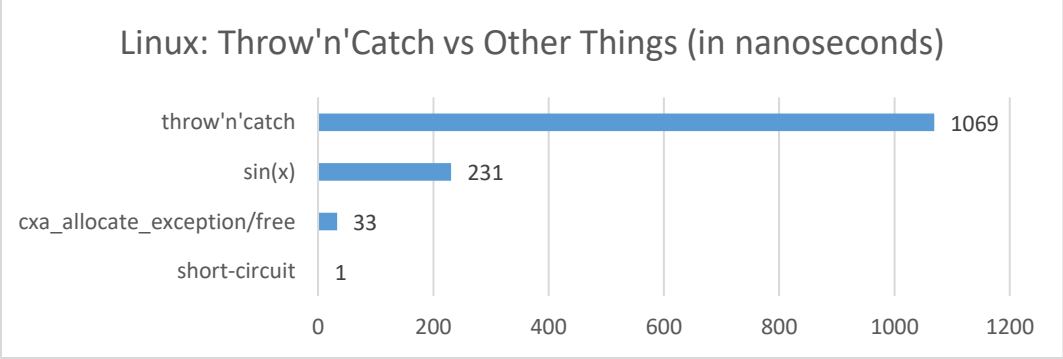
4 Simple throw and catch optimization

Finally, we will look at exception “short-circuiting” pattern, where after inlinining we end up with throw and catch of the same exception in the same function.

Before	After

`try {`
`  if (pred())`
`  throw cancelled{};`
`  may_throw();`
`} catch (cancelled const &) {`
`  payload();`
`} catch (something-else) {…} …`
`…`
	`try {`
	`  if (pred())`
	`  goto call_payload;`
	`  may_throw();`
	`} catch (cancelled const &) {`
	`  goto call_payload;`
	`} catch (something-else) {…} …`
	`goto payload_bypass;`
	
	call_payload: **payload();**
	payload_payload:
	…

This transformation results in 1000+ times speedup on Linux amd64 for synthetic benchmark that runs very inexpensive cleanup (an increment of a variable) and a predicate that always returns true. For comparison, we list per iteration cost of a throw and catch, computing sine function of a double, performing an allocation and deallocation of memory for an exception using Itanium ABI’s cxa_allocate_exception/cxa_free_exception and finally a direct call to payload if predicate is true (short-circuit label).



This pattern may occur after inlining in asynchronous generator coroutines that use cancelled exception to indicate the need to stop execution of the asynchronous generator in the presence of Async RAII [D1662R0].

5 Complications

So far, we considered the code where optimizer knows that none of the functions executed during cleanup or catch handling that is replaced with regular control flow contains (directly or indirectly) calls to std::current_exception(), std::uncaught_exceptions() or throw; statement.

When such functions calls are encountered, the optimizer would need to insert calls to the exception runtime to appropriately increment and decrement the uncaught_exceptions count and set up/tear down current exception. While it will slow down the execution relative to the regular control flow (as we have seen in the previous

section to allocate and free memory for an exception takes about 33ns), it is still significantly cheaper than invoking full exception propagation machinery (~1000ns).

Note: that setting the current exception is needed even if unknown functions calls are only in the cleanup code leading towards the catch. This is needed because a user can replace a standard terminate handler with his own and observe the current exception in the replacement terminate handler either via throw; or a call to std::current_exception().

6 Scope Guard interactions

While previous section describes a general way of dealing with calls to functions that may call std::current_exception() and friends. There are cases where calls to the exception runtime are not needed. Consider the upcoming scope guard facility [P0052] that is built on top of uncaught_exceptions() function.

User writes	Optimizer sees

`auto _= std::make_scope_fail([]{`
`  payload();`
`});`
`may_throw();`
	`int tmp = uncaught_exceptions();`
	`try { may_throw(); }`
	`on_unwind`[4] `{`
	`  if (tmp < std::uncaught_exceptions())`
	`  payload();`
	`  // exception unwind continues`
	`}`
	`if (tmp < std::uncaught_exceptions())`
	`  payload();`

If an optimizer is taught the semantics of the uncaught_exceptions function, it can remove uncaught_exception away in all cases covered by ScopeGuard paper [P0052], assuming that destructors for helper temporary objects are inlined and calls to uncaught_exceptions are observable:

Before	After

`int tmp = uncaught_exceptions();`
`try { may_throw(); }`
`on_unwind {`
`  if (tmp < std::uncaught_exceptions())`
`  payload();`
`  // exception unwind continues`
`}`
`if (tmp < std::uncaught_exceptions())`
`  payload();`
`}`
	`try { may_throw(); }`
	`on_unwind { payload(); }`

This and similar optimizations of std::uncaught_exceptions() will increase the reach of exception short-circuiting optimizations described earlier.

[4] Here on_unwind marks a block of code to execute during exception unwinding. This capability is present in the intermediate representation of a compiler, but, does not have a direct language equivalent in C++. ScopeGuard tries to emulate it.

7 Further Simpifications

We expect that further optimizations are possible with comparable complexity of the implementation that would handle more patterns. For example, we can add an optimization of the pattern occurred after inlining of Lippincott Function [LippFunc].

Before	After

`try {`
`  may_throw();`
`} catch (...) {`
`  try { throw; }`
`  catch (A) { handler1(); }`
`  catch (B) { handler2(); }`
`  catch (...) { handler3(); }`
`}`
	`try {`
	`  may_throw();`
	`}`
	`catch (A) { handler1(); }`
	`catch (B) { handler2(); }`
	`catch (...) { handler3(); }`

While the original motivation for exploring optimization of these patterns was that they frequently occur in coroutines, these optimizations may be beneficial for other application categories and more optimizations of this kind can be developed.

8 Acknowledgements

Many thanks to those who reviewed the drafts of this paper and provided valuable feedback, among them: Lewis Baker, Neeraj Singh, Modi Mo, Billy O'Neil, Gabriel Dos Reis.

9 Bibliography

[D1662R0] Lewis Baker. "Adding async RAII support to coroutines" (WG21 paper pre-publication draft, 2019-06-10).

[N2952] V. Voutilainen. "Accessing current exception during unwinding" (WG21 paper, 2009-09-21).

[P0052] Peter Sommerlad, Andrew L. Sandoval. "Generic Scope Guard and RAII Wrapper for the Standard Library" (WG21 paper, 2019-02-19).

[N4152] Herb Sutter. "uncaught_exceptions()" (WG21 paper, 2014-09-30).

[LippFunc] C++ Secrets. "Using a Lippincott Function for Centralized Exception Handling" (WG21 paper, 2013-12-13).

10 Appendix

Measurements were done on a Lenovo P50 with Skylake Xeon(R) CPU E3-1505M v5 3.7Ghz CPU running Ubuntu 18.04 LTS using modified version of clang compiler that included eh optimization pass.

https://reviews.llvm.org/D63388 - section 3 and 4 optimizations

https://reviews.llvm.org/D55186 - section 2 catch and rethrow optimization