P2477R3: Allow programmers to control coroutine elision

Chuanqi Xu, 23/2/8

Introduction & Motivation

```
CoroTask bar();
CoroTask foo() {
   co_return co_await bar();
}
```

```
CoroTask bar();
CoroTask foo() {
   void *FramePtrOfBar = malloc(size needed);
   Construct Frame (Including Construct promise of bar)
   auto T = <promise-of-bar>.get_return_object();
   // assume no other operations for simplicity
   co_return co_await T;
}
```

Every time a coroutine get called, one dynamic allocation is made.

(Let's forget the wording actually says 'An implementation **may** need to allocate additional storage' now)

Introduction & Motivation

Dynamic allocation is expensive and it should be extremely good to avoid it. So Richard and Gor designed HALO to eliminate the dynamic allocation in P0981R0. HALO is called coroutine elision nowadays. After coroutine elision is made, the code might look like:

```
CoroTask bar();
CoroTask foo() {
    co_return co_await bar();
}
```

```
CoroTask bar();
CoroTask foo() {
    FrameTypeOfBar FrameOfBar;
    Construct Frame (Including Construct promise of bar)
    auto T = <promise-of-bar>.get_return_object();
    // assume no other operations for simplicity
    co_return co_await T;
}
```

Coroutine elision replaces the dynamic allocaton with a local variable simply.
Now the frame of bar() is a local variable of foo().

Introduction & Motivation

But coroutine elision is not satisfying:

- It couldn't optimize many cases. Now it could only optimize simple synchronized things (generators, tasks, etc). But coroutine outperforms in asynchronized situations.
  - The compiler is very hard to tell if it is safe to do coroutine elision.
  - The compiler gives up many cases to avoid miscompile.
- It is not well defined.
  - The library writers don't know why it work and why not.
  - The code worked now might be unavailable in other version of the same compiler.
- It is not implemented by every compilers: https://godbolt.org/z/hs6szacbe

Introduction & Motivation

We could solve the 2 problems by standardizing coroutine elision:

- Now programmers know how to make their coroutines elidable.
- Programmers know more about whether it is safe or not to do elision.

Introduction & Motivation

Dynamic allocation is expensive and unnecessary in many situations. However, programmers lack a well-defined method to avoid it. I feel it is not consistent with zero abstract rule. So we should provide one.

Proposed Solution

```
enum class coro_elision { may, always, never };

struct promise_type {
  static constexpr coro_elision  must_elide(P0, P1, …, P_n) {
      …
  }
}
```

For each call for a coroutine, the compiler would try to evaluate `promise_type::must_elide()` at constexpr time.

- If the result is `coro_elision::always`, the call is guaranteed to do coroutine elision.
- If the result is ` coro_elision::never`, the call is guaranteed to not do coroutine elision.
- Otherwise (`promise_type::must_elide()` is not exist, we could not evaluate it at constexpr time or the result is `coro_elision::may`), it is up to the compiler to do coroutine elision or not. (As now)

Proposed Solution

```
struct TaskPromiseAlternative : public TaskPromiseBase {
    // Return the last argument if its type is std::coro_elision.
    // Return std::coro_elision::may otherwise.
    template <typename... Args>
    static constexpr std::coro_elision must_elide(Args&&... args) {
        using ArgsTuple = std::tuple<Args...>;
        constexpr std::size_t ArgsSize = std::tuple_size_v<ArgsTuple>;
        if constexpr (ArgsSize > 0) {
            using LastArgType = std::tuple_element_t<ArgsSize - 1, ArgsTuple>;
            if constexpr (std::is_convertible_v<LastArgType, std::coro_elision>)
                return std::get<ArgsSize - 1>(std::forward_as_tuple(args...));
        }

        return std::coro_elision::may;
    }
};
```

Examples

```
ControllableTask an_elide_controllable_coroutine(std::coro_elision elision_state = std::coro_elision::may) {
    co_return 43;
}

NormalTask example(int count) {
  int sum = 0;
  for (int i = 0; i < count; ++i) {
    auto t = an_elide_controllable_coroutine(std::coro_elision::always);
    sum += co_await std::move(t);
  }

  co_return sum;
}

NormalTask example2(int count) {
  std::vector<TaskBase *> tasks;
  for (int i = 0; i < count; ++i)
    // Or
    //    tasks.push_back(an_elide_controllable_coroutine());
    tasks.push_back(an_elide_controllable_coroutine(std::coro_elision::never));

  co_await whenAll(tasks);
  // ...
}
```

Limitations

To do coroutine elision, we need to know the callee body at compile time. It means that, we couldn’t do coroutine elision even if `promise_type::must_elide()` evaluates to `coro_elision::always` in following cases:

- The callee definition is unreachable to the point of the call. For example, the callee is defined in another normal translation unit.
- The call is an indirect call. For example: virtual functions and function pointers.

There shouldn’t be any solution for these 2 limitations.

Concerns

I received 2 main concerns for this proposal:

- The potential misuse.
- The wording.

Concerns - misuse

What if `promise_type::must_elide()` evaluates to `coro_elision::always` when it is not safe to do so?
- It is undefined behavior. Probably a segmentation fault in practice.

Concerns - misuse - example

What if `promise_type::must_elide()` evaluates to `coro_elision::always` when it is not safe to do so?
- It is undefined behavior. Probably a segmentation fault in practice.

```
Task<int> example2(int task_count) {
  std::vector<Task<int>> tasks;
  for (int i = 0; i < task_count; i++)
    // The coroutine state of coro_task() becomes a local
    // variable of the loop.
    tasks.emplace_back(coro_task(std::coro_elision::always));

  // The semantics of whenAll is that it would
  // complete after all the tasks are completed.
  //
  // The tasks would contain to dangling pointers only.
  co_return co_await whenAll(std::move(tasks));
}
```

Concerns - misuse

What if `promise_type::must_elide()` evaluates to `coro_elision::always` when it is not safe to do so?
- It is undefined behavior. Probably a segmentation fault in practice.

My thought: this is not a silver bullet which can solve everything perfectly. It requires the user to understand the lifetime model of the coroutine they are using. It is completely fine to not use it if the user has any concern. I think this is consistent with “pay for what you use”.

Concerns - wording

Another concern is if we can make the wording precisely and shortly.

I feel we can look into the problems after we move the paper to CWG.

Implementation

A demo implementation is available for clang.
For other compilers, from my personal experience, the difficulties to implement coroutine elision is about the analysis part. Since the paper move the job of safety analysis to programmers, it would be much easier for other compilers to implement it.

Conclusion

- Motivation: Dynamic allocation is expensive and unnecessary in many situations. However, programmers lack a well-defined method to avoid it. It violates zero abstract rule.
- Proposal:
  - Introduce static constexpr member function `promise_type::must_elide()` to allow programmer control coroutine elision by template metaprogramming.
- Limitation: We must see the coroutine body at constexpr time if we want to do coroutine elision.
  - This limits the use of virtual function, function pointers and calls to other source TUs.
- Concern:
  - Undefined behavior (SegFault generally in practice) if the users misuse the feature. This is the price we need to pay.
  - If we can make the wording precisely and shortly.
- Implementation: Yes.
- Examples: Yes.
- Wording: Not yet.

Thanks

Possible Q&A

Q: Would coroutine elision be affected by copy elision, or vice-versa?

```
template<typename... Args>
ElidedTask<void> foo_impl(Args... args) { co_await whatever(); ... }

template<typename... Args>
ElidedTask<void> foo(std::tuple<Args> t) {
  co_await std::apply([](Args&... args) { return foo_impl(args...); }, t);
}
```

A: The return object of `foo_impl()` would be copy elided. So the question is what about the elided frame? Would it lives in foo()? Or in the lambda?

The elided coroutine frame would live in the lambda sadly. So the code above is problematic.

Although the names of coroutine elision and copy elision look similar, they are two different and unrelated things.

Possible Q&A

Q: Why must_elide() must be a static constexpr member function?

A: Since we need to get the result before we construct promise_type. So it must be a static member function of promise_type.

But it could be a dynamic function actually. For example, we could generate following code:

```
1   coroutine_status_ty coroutine_status;
2   coroutine_handle<promise_type> *handle_;
3   if (promise_type::must_elide())
4       handle_ = &coroutine_status;
5   else
6       handle_ = promise_type::operator new(size_of(coroutine_status_ty));
7   // use of handle_
```

I just failed to find the use cases.
Note that we still need to see the coroutine body during compile time.

Possible Q&A

Q: Why compiler fails to do coroutine elision for asynchronous cases?

A: This relates to pointer analysis, which is internal to compiler.

Generally, we would submit a task to executor to resume a coroutine. And the submitted task contains the handle of awaiting coroutine generally. And from the perspective of compiler, it just find the coroutine handle is sent to other functions and the compiler don't know what would happen here. The compiler could only assume the handle is escaped so that the compiler would think the lifetime of the awaiting coroutine is leaked from the current function.

As a result, the compiler is failed to do coroutine elision for any asynchronous coroutine.

Possible Q&A

Q: Would the compiler do coroutine elision for synchronous coroutine?

A: It also depends on the pointer analysis, which is complex. The short answer here would be possible no for complex cases.