Allow programmer to control and detect coroutine elision

Chuanqi Xu, 2022/5/26

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.

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

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

Is it safe to do coroutine elision for the call `bar()` in `foo()`?

Introduction & Motivation

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

Is it safe to do coroutine elision for the call `bar()` in `foo()`?

Technically, no. Since we don't know what would happen if we `co_await` an AsyncTask.

Introduction & Motivation

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

Is it safe to do coroutine elision for the call `bar()` in `foo()`?

Technically, no. Since we don't know what would happen if we `co_await` an AsyncTask.

But it could be. As a library writer, I could be sure that `foo()` would be resumed only if bar() is completed. In other words, the lifetime of `bar()` is covered by the lifetime of `foo()`.
So it would be safe to make the frame of `bar()` to be a local variable of `foo()`.

Introduction & Motivation

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

The example shows that programmer knows more than the compiler on specific cases.

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() {
         …
   }
}
```

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`), the behavior is unspecified. (As now)

Proposed Solution

By the method, we could control the coroutine elision at callsite.

```cpp
struct ShouldElideTagT;
struct NoElideTagT;
struct MayElideTagT;

template<typename T>
concept ElideTag = (std::same_as<T, ShouldElideTagT> ||
                    std::same_as<T, NoElideTagT> ||
                    std::same_as<T, MayElideTagT>);

template <ElideTag Tag>
struct TaskPromiseAlternative : public TaskPromiseBase {
    static constexpr coro_elision must_elide() {
        if constexpr (std::is_same_v<Tag, ShouldElideTagT>)
            return coro_elision::always;
        else if constexpr (std::is_same_v<Tag, NoElideTagT>)
            return coro_elision::never;
        else
            return coro_elision::may;
    }
};

template <ElideTag Tag = MayElideTagT>
using AlternativeTask = Task<TaskPromiseAlternative<Tag>>;
using Task = AlternativeTask<MayElideTagT>;

template <ElideTag Tag = MayElideTagT>
AlternativeTask<Tag> alternative_task () { /* ... */ } // This is a coroutine

Task NaturalTask() { /* ... */ } // This is a coroutine.

int foo() {
  auto t1 = alternative_task<ShouldElideTagT>();
  // Task::elided would call coroutine_handle::elided
  assert (t1.elided());
  auto t2 = alternative_task<NoElideTagT>();
  assert (!t1.elided());
  // The default case, which would be general for most cases
  alternative_task();
  // The author of the function don't want us to control its elision
  // or the author think it doesn't matter to elide or not. After all,
  // the compiler would make the decision.
  NaturalTask();
}
```

Proposed Solution

If we could control the coroutine elision, it is odd that we couldn't observe it.

```
namespace std {
  template<>
  struct coroutine_handle<void>
  {
    // ...
    // [coroutine.handle.observers], observers
    constexpr explicit operator bool() const noexcept;
    bool done() const;
+   bool elided() const noexcept;
    // ...
  };

  template<class Promise>
  struct coroutine_handle
  {
    // ...
    // [coroutine.handle.observers], observers
    constexpr explicit operator bool() const noexcept;
    bool done() const;
+   bool elided() const noexcept;
        // ...
  };
}
```

Limitations & Concerns

Supportting `coroutine_handle<>::elide()` would require us to store information in coroutine frame. The coroutine frame would look like:

```
struct CoroutineFrame {
    void (*resume_func)(CoroutineFrame*);
    void (*destroy_func)(CoroutineFrame*);
    bool Elided;
    promise_type;
    // any other information needed
};
```

It would break the ABI.

Also the use cases are limited to test cases.

I am not sure if it is good to support it.

Limitations & Concerns

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 body is defined in unreachable TU.
- The call is an indirect call. For example: virtual functions and function pointers.

Limitations & Concerns

For the limitation to the unreachable TU, I think it is fine in practice since if you try to control coroutine elision in the proposed way, the coroutine should be a template function. And the template function should be defined in headers, current source file or in importable TUs.

From the perspective of wording, we could say:
    The definition of a coroutine with `promise_type::must_elide()` evaluated to `coro_elision::always` must be reachable to its use points.

From the perspective of implementation, We could ban this use during runtime by removing the call to allocation function if `promise_type::must_elide()` evaluates to `coro_elision::always`. Then it would throw exception if it fails to elide since it is in another TU.

Limitations & Concerns

For the limitation to indirect call, I think we could ban it directly:
    Coroutines with `promise_type::must_elide()` evaluated to `coro_elision::always` couldn't be virtual function or to take the address.

Limitations & Concerns

Another big concern raised by Mathias is the misuse.

```
Task<int> example2(int task_count) {
  std::vector<ElidedTask<int>> tasks;
  for (int i = 0; i < task_count; i++)
    tasks.emplace_back(coro_task());

  // the semantics of whenAll is that it would
  // complete after all the tasks completed.
  co_return co_await whenAll(std::move(tasks));
}
```

If the call to `coro_task()` would do coroutine elision all the time, the code would look like:

Limitations & Concerns

Coroutine elision would make the coroutine status a local variable in the enclosing block.

```
Task<int> example2(int task_count) {
  std::vector<coroutine_status_ty *> tasks;
  for (int i = 0; i < task_count; i++) {
    coroutine_status_ty coroutine_status;
    tasks.emplace_back(&coroutine_status);
  }

  // Now all the pointer in tasks are dangling
  // pointer!
  co_return co_await whenAll(std::move(tasks));
}
```

So the `tasks` stores all of dangling pointers!
The behavior of above code is undefined.

Limitations & Concerns

I think it is acceptable since:

- must_elide() is not designed as a silver bullet to solve all the problems.
- must_elide() is not a forced option.
- must_elide() tries to give the rights to the users and the user need to pay the risk (If they know clear enough about the lifetime model of the coroutine components they are using.)
- This is the price we need to pay. No matter what the solution is, as long as we want to control coroutine elision in the language side, we would always meet the problem.

Implementation & Examples

Implementation in clang: https://github.com/ChuanqiXu9/llvm-project/tree/CoroMustElide

Examples: https://github.com/ChuanqiXu9/llvm-project/tree/CoroMustElide/MustElide

Implementation in other compilers:

- I believe it wouldn't be hard. Since the hardest part of coroutine elision is the analysis part to decide whether or not it is safe to do the elision, the conversion part should be easy and trivial. The proposal tries to offload the analysis part to programmers so it wouldn't be hard for compilers to do conversion.

Wording

Not available wording yet.
We could start if we could get some consensus.

Conclusion

- Motivation: Dynamic allocation is expensive and unnecessary in many situations. However, programmers lack a well-defined method to avoid it. It looks to violate zero abstract rule.
- Proposal:
  - Introduce static constexpr member function `promise_type::must_elide()` to allow programmer control coroutine elision by template metaprogramming.
  - Introduce `coroutine_handle<>::elided()` to allow programmers to observe if it is elided. (Use cases other than test case?)
- Limitation: We must see the coroutine body if we want to do coroutine elision.
  - This limits the use of virtual function, function pointers and calls to other source TUs. We could do compile time check for the previous 2 cases and we could add a runtime check for the last case.
- Concern: Undefined behavior (SegFault generally in practice) if the users misuse the feature.
  - This is the price we need to pay.
- 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: Could we get the result elided() at constexpr time?

A: No. We need to store from the coroutine frame. Although there might be cases that the compiler could optimize it. For example:

```
__attribute__((noinline))
bool IsElided(coroutine_handle<> handle) {
  return handle.elided();
}
```

We have no way to get the result at compile time.

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: Could dynamic `must_elide()` solves the limitations that we need to see the callee function body at compile time?

A: It couldn't. Since coroutine elision would emit a local variable in the caller's stack frame but the size of the stack frame should be fixed after compile time. So the idea to do coroutine elision dynamically is basically conflicting.

Possible Q&A

Q: Why not an attribute like `always_coro_elide` (mimics `always_inline`)

A: (1) The `always_inline` attribute is not in standard first.
(2) Although `always_coro_elide` looks more fine-grained than the current proposal, the proposed solution is able to control the elision at callsite actually. So the current proposal is more fine-grained.

Possible Q&A

Q: Could we control coroutine elision at callsite in a simpler syntax?

A: We could make it greatly if we have attributes at expression level. Like:

```
1   Task bar();
2   Task foo() {
3       co_return co_await [[CoroElision]] bar();
4   }
```

But we don't support expression level attributes now. Another idea is to use statement level attribute to do coroutine elision for all the calls to coroutines in the statement:

```
1   Task bar();
2   Task bar2();
3   Task foo() {
4       [[CoroElision]] co_return co_await bar() +
5                                  co_await bar2();
6   }
```

It looks odd at first. And it looks easier to misuse if the statement is long.

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 the 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.