Document	P1663R0
Reply To	Lewis Baker <lbaker@fb.com>
Date	2019-06-18
Audience	Evolution

Supporting return-value optimisation in coroutines

Abstract

Note that this paper is intended to motivate the adoption of changes described in P1745R0 for C++20 to allow adding support for async RVO in a future version of C++. The changes described in this paper can be added incrementally post-C++20.

The current design of the coroutines language feature defines the lowering of a `co_await` expression such that the result of the expression is produced by a call to the `await_resume()` method on the awaiter object.

This design means that a producer that asynchronously produces a value needs to store the value somewhere before calling `.resume()` so that the `await_resume()` method can then return that value. The most common places the value is stored is either in the awaiter object itself or, if the value is being produced by a coroutine, then in the coroutine's promise object.

However, this means that the `await_resume()` method will need to either return a reference to this stored value, and the consumer's use of the result is then bound by the lifetime of the storage chosen by the awaitable, or `await_resume()` will need to copy/move the object into a new prvalue result.

This paper explores what a potential design for coroutines would look like that supported return-value-optimisation in such a way that this extra move was not necessary by allowing the async operation to construct the result of the `co_await` expression directly in-place instead of having to return the result from the `await_resume()` method.

This paper then looks at how this capability can be used to address the larger challenge of piping return-value-optimisation through a number of nested calls to coroutines so that the final result is constructed in-place without incurring a move operation at every call.

Finally, this paper proposes that changes to the design of `coroutine_handle` are made for C++20 to enable adding return-value optimisation in a future version of the standard. The proposed changes described in detail in P1745R0 - "Coroutine changes for C++20 and beyond".

The changes proposed in P1745R0 that would enable RVO to be added in a future version would also enable adding first-class support for async cancellation of a coroutine in a future version of the standard; this is important for supporting cancellation of an `async_generator` coroutine in the presence of async RAII (see [P1662R0] for details).

Background

The current design of the coroutines language feature defines a `co_await` expression to be translated as follows.

```
co_await <expr>
```

Is translated into something roughly equivalent to the following (some casts omitted for brevity)

```
auto&& __operand = <expr>;
auto&& __awaitable = maybe_await_transform(promise, __operand);
auto&& __awaiter = maybe_operator_co_await(__awaitable);
if (!__awaiter.await_ready()) {
  <suspend-coroutine>
  auto __h = coroutine_handle<promise_type>::from_promise(promise);
  /*tail*/ return __awaiter.await_suspend(__h).resume();
  <resume-point>
}
__awaiter.await_resume();
```

When a coroutine is resumed by calling `.resume()`, either explicitly or implicitly by returning a `coroutine_handle` from `await_suspend()`, this resumes the coroutine at `<resume-point>`. The coroutine then immediately calls `__awaiter.await_resume()` which produces the result of the `co_await` expression.

This design means that a producer that asynchronously produces a value needs to store the value somewhere before calling `.resume()` so that `__awaiter.await_resume()` can return that value. The most common places the value is stored is either in the `__awaiter` object itself or, if the value is being produced by a coroutine, then in the coroutine's promise object.

However, this means that the `await_resume()` method will need to either return a reference to this stored value, and the consumer's use of the result is then bound by the lifetime of the storage chosen by the awaitable, or `await_resume()` will need to copy/move the object into a new prvalue result so that the lifetime is then defined by the caller.

Regardless of which approach is taken, this design means that the library will typically require an extra move operation when the user writes '`auto x = co_await some_task();`' if that operation completes asynchronously.

Motivation

Consider a straight-forward implementation of a `task<T>` type that stores the return-value in the promise object of the coroutine. The awaiting coroutine suspends before resuming the task's coroutine and then when the task completes it the awaiting coroutine is resumed which then calls the `await_resume()` method which returns a copy of the return-value that is move-constructed from the value stored in the promise.

Example: A simple case of one task awaiting another task.

```
struct big_object {
  explicit big_object(int value);
  big_object(big_object&&);

  // Assume it has lots of data-members
  float data[100];
};

task<big_object> foo() {
  co_return big_object{x};
}

task<void> bar() {
  big_object obj = co_await foo();
  std::cout << std::ranges::accumulate(obj.data);
}
```

In this code snippet the control flow from the `co_return` statement proceeds as follows:

- A temporary `big_object` is constructed inside `foo()`.
  This should typically be placed on the stack, but compilers may allocate its storage in the coroutine frame (as Clang 8.0 does).
- A reference to this temporary object is passed to `foo()`'s `promise.return_value()` method.
- The `return_value()` method move-constructs the value into a data-member of the promise.
- After `return_value()` returns the program executes '`goto final_suspend;`' which exits scopes within `foo()` until `foo()`'s `final_suspend()` is reached.
  This destroys the temporary `big_object`.
- At `foo()`'s `final_suspend()` it performs a symmetric transfer to resume the `bar()` coroutine.
- The resumption of `bar()` calls `await_resume()` on the task's awaiter.
- The `await_resume()` method move-constructs its return value from the value stored in the promise. This return value is a temporary object.
- The return-value is stored in the variable '`obj`'.
  We can rely on the copy of the return-value into the variable '`obj`' being elided by the compiler, so the '`obj`' variable is essentially move-constructed directly from the object stored in the promise.

This chain of events means that the returned value ends up being move-constructed twice before it ends up in the resulting '`obj`' variable. Compare this to the equivalent synchronous code, which, under C++17, guarantees that the variable is constructed in-place from the return-value and that the move-constructor is not called.

These **extra calls to move-constructors** can potentially be expensive for some types (eg. types with a large number of data members) and it may be important for some use-cases to reduce the number of calls to the move-constructor to one or ideally to zero.

The fact that the object must be move-constructed means that `task<T>` types are **limited to only working with movable return-types**. It is not possible for such a coroutine to return a non-movable type that is constructed in-place, whereas this is something you can do with ordinary functions since C++17 due to guaranteed copy-elision.

Further, if we consider a call-chain where a coroutine calls another and immediately returns the return-value of the nested coroutine. Then the **storage required for the coroutine grows by the size of the return-value for each nested coroutine** that returns the value. As coroutine frames are typically heap-allocated, this can result in larger coroutine frame heap-allocations.

For example: Nested calls to coroutines in tail-position.

```
struct big_object {
  big_object(int a, int b);
  float data[1000];
};

task<big_object> get_big_object1(int a, int b) {
  co_return big_object{a, b};
}

task<big_object> get_big_object2(int a) {
  co_return co_await get_big_object1(a, 42);
}

task<big_object> get_big_object3() {
  co_return co_await get_big_object2(123);
}
```

Let's analyse the coroutine frame sizes here:

- The coroutine frame for `get_big_object1()` contains the task's promise object which holds storage for the return-value (size ~4000 bytes).
- The coroutine frame for `get_big_object2()`, assuming the allocation of `get_big_object1()`'s coroutine frame is inlined[1] into the frame of `get_big_object2()`, will be the size of `get_big_object1()` (~4000 bytes) + the size of the promise (~4000 bytes). This means that the resulting coroutine frame is ~8000 bytes.
- Similarly, the coroutine frame for `get_big_object3()` will be the size of `get_big_object2()` frame plus the size of the promise for `get_big_object3()`. This results in a coroutine frame of ~12000 bytes.

This growth of the coroutine frame size by the size of the return-type for each nested call means that factoring code out into smaller coroutine functions is often not zero overhead.

If we were instead able to directly construct the return-value in its final place then we could reduce the storage required for return-values from O(call-depth) to O(1). This would also reduce the number of calls to the move-constructor from O(call-depth) to zero.

---

[1] See P0981R0 - "Halo: coroutine Heap Allocation eLision Optimization"

There are multiple aspects to this problem and it will require multiple pieces of the puzzle to reach this goal.

This can be roughly broken down into the following pieces:

1. Constructing the result of a `co_await` expression in-place.
2. Avoiding the construction of the initial temporary object in the wrong place.
3. Piping the return-address through nested calls.

The first item requires language support and is the main focus of this paper.

The second item can be addressed with a library solution if we require that the user explicitly opt-in to copy-elision when writing the `co_return` statement.
eg. '`co_return std::in_place_construct(args...);`'
or '`co_return std::in_place_construct_type<T>(args...);`'

However, we would require some additional language changes on top of those proposed in P1745R0 to allow eliminating the copy using the natural return-syntax. ie. '`co_return T{args...};`'. It is expected that these extensions could be added incrementally post-C++20.

The third item should be solvable using a purely library solution, but may depend on some other enhancements to coroutines, such as P1713R0 which relaxes some restrictions on `co_return`, or support for making `co_return` a suspend-point.

RVO for coroutines and co_await expressions

Return-Value Optimisation for normal functions

Let's first start with a recap of how ordinary functions implement return-value-optimisation.

When a caller invokes a function that makes use of return-value-optimisation (ie. where the return-value is not trivially movable and destructible and cannot be returned in a register) then the caller typically reserves some storage for the return value in its local stack-frame and then passes a pointer to this storage as a hidden parameter to the function being called.

The function being called can then construct the return-value directly in-place at the address provided in the hidden parameter.

Since C++17, ordinary functions guarantee that a copy of the return-value can be elided if you return a prvalue. i.e. that the return value is constructed directly in-place.

There are a few other variations on return-value optimisation:

Transitive Return-Value Optimisation

If a function is returning the result of another function call then the function can just pass the address of its return-value as the address of the called function's return value so that the called function constructs the result in-place at the return-value address provided by the caller.

```
T foo() {
  // In-place constructs T{} at the address of 'result'
  return T{};
}

T bar() {
  // Passes address of 'result' as return-value address to 'foo()'
  return foo();
}

void baz() {
  // Passes address of 'result' as return-value address to bar()
  T result = bar();
}
```

Named Return-Value Optimisation

If a function returns the same named local variable on all code-paths and the type of this variable matches the type of the return-value then the compiler is free to construct that named variable in-place at the return-value address and elide the copy of the local variable into the return-value.

https://wandbox.org/permlink/xoRkOX6rtpHuBA2O

```
T callee() {
  T value;
  printf("&value == %p\n", (void*)&value);
  return value;
}

void caller() {
  T result = callee();
  printf("&result == %p\n", (void*)&result);
}
```

This is allowed to output the following:

```
&value == 0x7ffeef0cb350
&result == 0x7ffeef0cb350
```

Return-Value Optimisation for `co_await` expressions

Status Quo

With the current specification of coroutines, the result of a `co_await` expression is produced by calling the `await_resume()` function. This can either return a value or can throw an exception.

A coroutine reserves storage for the result of a `co_await` expression either on the stack or in the coroutine frame based on whether or not the result spans a suspend-point. The address of the storage is logically passed into the call to the `await_resume()` method as the return-value address, assuming the compiler applies return-value optimisation to this call.

However, for cases where a value is produced asynchronously, an Awaiter object will typically need to construct the result somewhere in a temporary location, often within the Awaiter object itself, before then resuming the coroutine with a call to `h.resume()` on the coroutine handle. The resumed coroutine will then immediately call `await_resume()` and use the result of that call as the result of the `co_await` expression. As the address of the result of the `co_await` expression cannot be known until the `await_resume()` method is called and `await_resume()` is only called after the coroutine is resumed this essentially forces this copy to be made.

There are some tricks that can be performed to eliminate this copy in some limited cases by storing the arguments to the return-value constructor in the awaiter and deferring the call to the return-value constructor until await_suspend(), but this is not a general solution and can quickly get complicated/expensive if the result can potentially be constructed via different constructor overloads.

Passing a handle to the return-value slot

To be able to implement return-value optimisation for a `co_await` expression we need instead to have the compiler somehow pass knowledge of the address of the result of the `co_await` expression into the Awaiter, rather than implicitly to `await_resume()`, so that it can directly construct the result in the correct location before resuming the coroutine.

The obvious approach here would be to incorporate the address of the return-value into the `coroutine_handle` passed to `await_suspend()` and allow constructing the return-value in-place by calling a method on the handle.
Instead of calling `handle.resume()` to resume the coroutine we could call `handle.set_value<T>(ctorArgs...)` to construct the result of the `co_await` expression in-place and then resume it.

Once we do this, however, it no longer becomes necessary to call `await_resume()` when the coroutine resumes to produce the result of the `co_await` expression as the result will have already

been constructed. Thus we can remove the `await_resume()` method from the revised Awaiter concept.

Once we remove the `await_resume()` method it is no longer possible for the coroutine to "pull" the result from the awaitable synchronously in the case that `await_ready()` returns `true`. We now need to always call `await_suspend()` to pass it the handle containing the return-value address so that it can construct the result. This means we can now also remove the `await_ready()` method from the revised Awaiter concept, leaving it with just the `await_suspend()` method.

Similarly, we can now remove the variant of `await_suspend()` that returns a `bool` as it no longer makes sense to resume the coroutine by returning false without first constructing the return value.

Finally, the `void`-returning `await_suspend()` method can now potentially be replaced by having it return `std::noop_coroutine()` instead of returning `void`.

This leaves us with a revised Awaiter concept that just has a single `await_suspend()` method that accepts and returns a `coroutine_handle`.

Example: Simple Awaitable that is always ready and returns the stored value.

Current Design	With support for RVO

`template<typename T>`
`struct just {`
`  T value;`
 	
`  bool await_ready() {`
`  return true;`
`  }`
 	
`  void await_suspend(coroutine_handle<>) {`
`  assert(false);`
`  }`
 	
`  T await_resume() {`
`  return value;`
`  }`
`};`
	`template<typename T>`
	`struct just {`
	`  T value;`
	
	`  template<typename Handle>`
	`  auto await_suspend(Handle h) {`
	`  h.set_value<T>(value);`
	`  return std::noop_coroutine();`
	`  }`
	`};`

Resuming with an exception

One implication of the elimination of the `await_resume()` method is that we no longer have code that runs inside the context of the coroutine after it resumes that is under the control of the awaitable. When we call `set_value()` to construct the result of the `co_await` expression and then subsequently resume the coroutine it will resume without throwing an exception.

This means there is nowhere to place a `throw` statement to allow throwing an exception from the `co_await` expression.

To support throwing an exception from a `co_await` expression, we would need to add another method to the `coroutine_handle`, say `set_exception(std::exception_ptr)` that would store the `exception_ptr` in the coroutine frame as a temporary and, when the coroutine is resumed, immediately rethrow the exception by calling `std::rethrow_exception()` with that stored exception.

Multiple Resumption Paths and Symmetric Transfer

Now that we've added a `set_exception()` method that resumes the coroutine along a code-path that throws an exception in addition to a `set_value()` method that resumes the coroutine along the normal-control flow path it raises the question of how we tell the compiler which of these paths to resume on when we return the `coroutine_handle` from the `await_suspend()` method.

Previously, when there was only a single path that the coroutine could be resumed on we could just return the `coroutine_handle` from `await_suspend()` and let the compiler tail-call the `resume()` method on the handle. However, now that there are two possible methods to resume the coroutine we need some other way to communicate which path the coroutine should be resumed on. Also, as these methods may require arguments to be passed we would need some way to tell the compiler which arguments to invoke the method with.

The paper P1662R0 also runs into the same issue with multiple resumption paths (in that case a choice between `set_done()` and `resume()`) and discusses some possible options for this that were ultimately discarded in the section titled "Challenges with this design" which I will refer the reader to and avoid duplicating here.

The approach that P1745R0 proposes to solve this problem is to separate the responsibilities of the `coroutine_handle` type into a "**suspend-point handle**", which represents a coroutine suspended at a particular suspend-point that could have multiple possible resumption paths, and a "**continuation handle**", which represents the selected resumption path.

With this design, the coroutine would pass a suspend-point handle as the parameter to the `await_suspend()` method and the `await_suspend()` method would return a continuation handle to symmetrically-transfer to.

The suspend-point handle passed to the `await_suspend()` method for an async-RVO awaitable would have a `set_value<T>()` method, which constructs the result in-place in the associated coroutine-frame, and a `set_exception()` method, which stores the `exception_ptr` in the associated coroutine-frame.

Both methods return a continuation handle that represents that particular chosen resumption path. The continuation handle can be invoked asymmetrically by calling `operator()` or symmetrically by returning it from an `await_suspend()` method.

Simplifying the Awaiter concept

Since there is now just a single method on the Awaiter, we can also consider merging the Awaiter concept together with the Awaitable concept. This would involve merging `await_suspend()` with `operator co_await()` by having the compiler pass the handle to `operator co_await()` instead.

Aside: There are some other benefits to doing this when combined with deferred coroutine frame creation (see [P1342R0]) as it gives access to the promise type of the caller prior to creating the coroutine frame and deciding on the `promise_type` for the callee. This would allow context to be propagated transparently from caller to callee.

Note that we still need to support `operator co_await()` being able to return temporary objects that are placed in the coroutine frame as this allows implementations to avoid heap allocations. So `operator co_await()` would be defined such that it either must return a `continuation_handle` or return an object with an `operator co_await()` defined, in which case the compiler would generate a chained call to its `operator co_await()`.

So to implement the Awaitable concept a type would now only need to implement the `operator co_await()` method. This method would return either a continuation handle or return an object that itself had an `operator co_await()` method that returns an Awaitable.

```
class schedule_op {
public:
  auto operator co_await() const {
    class awaiter : schedule_base {
    public:
      using schedule_base::schedule_base;

      bool await_ready() { return false; }

      void await_suspend(coroutine_handle<> h) {
        coro_ = h;
        this->tp_->enqueue(this);
      }

      void await_resume() {}

    private:
      void execute() noexcept override {
        coro_.resume();
      }

      coroutine_handle<> coro_;
    };
```

```
class schedule_op {
public:
  template<typename Handle>
  auto operator co_await(Handle) {
    class awaiter : schedule_base {
    public:
      using schedule_base::schedule_base;

      auto operator co_await(Handle h) {
        sp_ = h;
        this->tp_->enqueue(this);
        return std::noop_continuation();
      }

    private:
      void execute() noexcept override {
        sp_.set_value<void>()();
      }

      // Store non-type-erased handle.
      Handle sp_;
    };

    return awaiter{tp_};
```

```
    return awaiter{tp_};
  }

private:
  thread_pool* tp_;
};
```

```
  }

private;
  thread_pool* tp_;
};
```

Specifying the return-type of an awaitable

One of the implications of removing the `await_resume()` method is that we no longer have any way to deduce what the result-type of a `co_await` expression will be as we won't know what type will be specified in the call to `handle.set_value<T>()` until `operator co_await()` is called.

This is similar to the problem faced by Senders in the Sender/Receiver design [P1341R0] where the base Sender concept does not report the set of types it is going to send but instead just constrains the receiver passed to submit() by checking that it is invocable with the types it is going to send.

The solution there was to introduce a refinement of the Sender concept, called a TypedSender, that contains a nested type-alias that can be queried to determine the type(s) with which the Receiver would be invoked.

We can do something similar for Awaitable types and require that you provide a nested `Awaitable::await_result_type` typedef which the compiler can inspect to determine what the type of the expression will be.

The compiler can then pass a suspend-point handle that has an overload of `set_value<T>()` where `T` is constrained to be the same as `await_result_type`.

Calculating the address of the return-value

The suspend-point handle that the compiler ends up passing to the `operator co_await()` method needs to be able to obtain the address of the return-value if it is to be able to construct the result of the `co_await` expression in-place.

To do this it needs to be able to obtain the offset of the result within the coroutine frame, however the offset of the result will not be known until much later in the compilation phase once the coroutine frame layout has been calculated.

However, as the compiler has complete control over generation of the suspend-point handle type it can insert some intrinsics into the body of the `set_value()` method that act as placeholders for the address calculation.

For example:

```
// This handle is instantiated with a reference to the coroutine function
// and the index of the suspend-point within this coroutine.
template<auto Coroutine, int SuspendPoint>
struct __suspend_point_handle {
  template<typename T>
  struct value_continuation_handle {
    // Allocate a unique integer within the specified Coroutine
    // This effectively becomes the index into a big switch that jumps to the
    // resumption point of the coroutine.
    static constexpr int __continuation_index = __builtin_unique_integer(&Coroutine);

    void* framePtr;

    // Invoke the continuation
    void operator()() {
      __builtin_coro_resume(&Coroutine, __continuation_index, framePtr);
    }
  };

  void* framePtr;

  template<typename T, typename... Args>
  value_continuation_handle<T> set_value(Args&&... args) {
    constexpr auto index = value_continuation_handle<T>::__continuation_index;
    void* address = __builtin_coro_return_address(&Coroutine, index, framePtr);
    new (address) T(static_cast<Args&&>(args)...);
    return value_continuation_handle<T>{framePtr};
  }

  // ... etc. for other members
};
```

e.g. An intrinsic `__builtin_coro_offset(&foo, index, framePtr)` could be used to represent the address of the return-value for the continuation with specified `index` in coroutine `foo`.
A similar intrinsic could be inserted within the body of the coroutine that keeps track of the final address of the local memory allocated for the return-value slot within the coroutine frame.

Once the coroutine frame layout has been calculated, a subsequent pass can lower `__builtin_coro_offset()` intrinsics to a constant offset calculation relative to the framePtr.

Benefits and Implications of RVO-style awaitables

The primary goal of async RVO is eliminating a copy when producing the result of the co_await expression. However, there are a number of other advantages and some potential implications of this design with regards to performance.

It can reduce the number of branches needed to extract the value
With the current design of awaitables with `await_resume()`, if an operation can potentially complete with either a value or an exception then the `await_resume()` method will typically need to read some

state which indicates whether it is a value or error and then branch either to the value-returning path or to the exception-throwing path.

For example: An awaitable that can either return a T or throw an exception.

```
struct my_awaitable {
  std::variant<std::monostate, T, std::exception_ptr> result_;
  std::coroutine_handle<> coro_;

  bool await_ready() { return false; }

  void await_suspend(coroutine_handle<> h);

  T await_resume() {
    if (result_.index() == 2) {
      std::rethrow_exception(std::get<2>(std::move(result_)));
    }
    return std::get<2>(std::move(result_));
  }
};
```

With the async RVO model the producer selects which resumption path to take and can directly jump to that resumption path, avoiding the need for an additional branch after the coroutine is resumed.

We can no longer run any awaitable-specific logic after the coroutine resumes
With this model, once the coroutine is resumed from a suspend-point the `co_await` expression is over and the awaitable can no longer affect the result of the coroutine. The only way to process the result of a `co_await` expression is to place that logic in the continuation of the coroutine.

Logic that a given awaitable previously placed inside await_resume() now just needs to be executed by the awaitable prior to resuming the coroutine. eg. restoring thread-local sate, transforming results, etc.

However, this means that algorithms that previously adapted other awaitables by implementing `await_resume()` to call the inner awaitable's `await_resume()` method to obtain the result and then transform result will now need to be implemented as a coroutine.

For example: We can explicitly adapt the existing Coroutines TS Awaitable state-machine to avoid allocations. With the async RVO-style Awaitables we would need to use a coroutine.

```
template<typename Inner, typename Func>
struct transform_awaitable {
  Inner inner;
  Func func;

  bool await_ready() {
    return inner.await_ready();
  }

  template<typename Handle>
  auto await_suspend(Handle h) {
    return inner.await_suspend(h);
  }
```

```
template<typename Inner, typename Func>
auto transform(Inner inner, Func func)
  -> task<std::invoke_result_t<
       Func, std::await_result_t<Inner>>> {
  co_return func(co_await inner);
}
```

```
  decltype(auto) await_resume() {
    return func(inner.await_resume());
  }
};

template<typename Inner, typename Func>
auto transform(Inner inner, Func func) {
  return transform_awaitable<Inner, Func>{
    std::move(inner), std::move(func)};
}
```

Note that we can also write the same code as the RHS under the Coroutines TS. The main rationale for for hand-writing this algorithm as an Awaitable instead of as a coroutine is to avoid the potential for introducing an extra implicit heap allocation for the allocation of a new coroutine frame. Although, the coroutine-based code is arguably much simpler and is also much more flexible. A coroutine can be extended easily to composing multiple child async operations whereas the manual approach can only adapt the logic for a single suspend-point.

It is possible that this concern about implicit heap allocation can be mitigated through additional changes to coroutines described in P1342R0 which discusses adding support for deferring the creation of the coroutine frame and representing coroutine frames as types that can be deterministically stack-allocated.

Also, the addition of non-type-erased coroutine handles proposed in P1745R0 in combination with non-type-erased coroutines described in P1342R0 should be able to achieve the equivalent level of optimisation available with the current Awaitable design.

No more built-in short-circuiting behaviour

The existing Awaitable design has a built-in short-circuiting behaviour that allows the coroutine to avoid suspending in the case that the result is available synchronously - in which case the result is then immediately pulled by calling await_resume().

With the Coroutines TS design, there is some overhead to suspending a coroutine - it needs to spill any values currently stored in registers into the coroutine frame and store the resumption state into the coroutine frame which indicates where the coroutine is to be resumed. So avoiding doing these operations can be a performance win.

With the async RVO model we no longer have this built-in short-circuiting behaviour. The coroutine will always suspend and call the `operator co_await()` method. However, with strongly-typed suspend-point handles which encode knowledge of the suspend-point in the type we can avoid the need to store the resumption state in the coroutine frame.

For example: A simple awaitable that conditionally suspends only

```
class async_event {
```

```
class async_event {
```

```
  std::atomic<bool> ready_ = false;
  std::coroutine_handle<> continuation_;
public:
  void set() {
    if (ready_.exchange(true)) {
      continuation_.resume();
    }
  }

  bool await_ready() {
    return ready_.load();
  }

  bool await_suspend(
      std::coroutine_handle<> h) {
    continuation_ = h;
    return !ready_.exchange(true);
  }

  void await_resume() {}
};
```

```
  std::atomic<bool> ready_ = false;
  std::continuation_handle continuation_;
public:
  void set() {
    if (!ready_.exchange(true)) {
      continuation_.resume();
    }
  }

  template<typename Handle>
  continuation_handle
  operator co_await(Handle sp) {
    auto continuation = sp.set_value<void>();
    if (!ready_.load()) {
      // Store type-erased resumption state
      continuation_ = continuation;
      if (!ready_.exchange(true)) {
        // Return to resumer
        return noop_continuation();
      }
    }
    // Otherwise return the continuation to
    // immediately resume it.
    return continuation;
  }
};
```

With this example, the state-machine that is built-in to the `co_await` expression for short-circuiting is manually implemented in the `operator co_await()` under the async-RVO model. Resumption state for the resumption of the coroutine is only written to memory if the short-circuit path is not taken. In the case the short-circuit path is taken, the continuation handle returned from operator can be returned in registers and immediately invoked by the coroutine machinery without needing to store the resumption state in the coroutine frame.

If the compiler is able to inline the `operator co_await()` body into the coroutine body then the performance and overhead is expected to be equivalent between the two approaches.

Generalising set_exception() to set_error<E>()

One of the possible extensions to the async RVO model is to allow the caller to specify the type of the exception to throw, rather than having to pass an exception_ptr which requires type-erasing the exception.

For example, a call to `handle.set_error<E>(ctorArgs...)` would construct in the coroutine frame a temporary object, `error`, of type `E`, and return a continuation-handle that when invoked will resume the continuation and immediately execute '`throw error;`'.

This can potentially allow the coroutine body to optimise the exception-handling more efficiently. eg. it may be possible to optimise the '`throw error;`' into the equivalent of a '`goto handler;`' in some cases.

See P1676R0 for the results of some experiments optimising exceptions.

Note that there may be several possible overloads of `set_error<E>()` called and the compiler would need to instantiate a different '`throw error;`' continuation for each invocation. This requires solving similar challenges to that of supporting multiple `set_value<T>()` overloads, as the compiler needs to be able to determine the set of types that it needs to instantiate the `set_error<E>()` continuation with. However, the challenges with `set_error<E>()` are reduced compared to `set_value<T>()` as it is not possible to have any subsequent expressions in the coroutine dependent on the concrete type of the exception thrown.

A transition path could be provided that exposed the `set_error<E>()` method on suspend-point handles but that initially had an implementation that called `set_exception()` with the result of a call to `std::make_exception_ptr(E(args...))`.

Avoiding copies when returning a temporary

The above changes avoid needing to store a copy of the value in the promise, reducing the number of move/copy operations from 2 to 1.

However, if we are returning a temporary object we still need to construct that value in the callee and then copy it to the caller by calling `set_value()` on the handle.

For example, a task implementation that uses async RVO would have a promise_type::return_value() method that looks like the following (using syntax from P1745R0):

```
template<typename T>
class task {
public:
  class promise_type {
    std::suspend_point_handle<std::with_value<T>, std::with_exception> consumer_;
    std::continuation_handle continuation_;

    template<ConvertibleTo<T> U>
    void return_value(U&& value) {
      continuation_ = consumer_.set_value<T>((U&&)value);
    }

    void unhandled_exception() {
      continuation_ = consumer_.set_exception(std::current_exception());
    }

    auto done() { return continuation_; }

    // ... rest of the definition omitted for brevity.
  };

  using handle_t = std::suspend_point_handle<
    std::with_resume, std::with_destroy, std::with_promise<promise_type>>;

  handle_t coro_;
```

```
  explicit task(handle_t coro) : coro_(coro) {}

  // ... other members omitted for brevity

  template<typename Handle>
  auto operator co_await(Handle handle) {
    coro_.promise().consumer_ = handle;
    return coro_.resume();
  }
};
```

Then when we write the following coroutine:

```
task<some_type> example() {
  int x = co_await something();
  co_return some_type{x};
}
```

The `co_return` statement is lowered as follows:

```
promise.return_value(some_type{x});
goto done;
```

A temporary instance of some_type is created as a local variable and a reference to this is passed into the return_value() method. This then calls sp_.set_value<T>() which move-constructs the result of the `co_await` expression.

What we would like to be able to do is avoid allocating this temporary as a local variable and elide the copy.

A library solution to avoiding construction of temporaries

To be able to avoid the construction of the temporary we need to be able to in-place construct the result at the return-value address. This requires deferring construction of the object until the call to set_value() so that the call to set_value() can directly construct the result.

To do this we can create a helper type that captures the arguments to the constructor and pass this into return_value() which can then pass this onto a new `set_value_from()` call which accepts this helper type.

```
template<typename... Args>
class in_place_construct {
public:
  in_place_construct(Args&&... args) noexcept : args(args...) {}

  template<typename T>
    requires Constructible<T, Args...>
  T* construct(void* ptr) {
```

```
    return std::apply([ptr](Args&&... args) {
      return ::new (ptr) T((Args&&)args...);
    }, args);
  }

private:
  std::tuple<Args&&...> args;
};

template<typename... Args>
in_place_construct(Args&&...) -> in_place_construct<Args...>;
```

Then we can define the new set_value_from() method on the suspend-point handle to call the construct() method on the inplace_construct object..

```
struct suspend_point_handle
{
  template<typename T, typename... Args>
    requires Constructible<T, Args...>
  value_continuation_handle<T>
  set_value_from(in_place_construct<Args...> inplace) {
    void* returnValueAddress = /* builtin magic here */;

    // Directly construct the result at the return-value address.
    inplace.construct<T>(returnValueAddress);

    return value_continuation_handle<value_type>{...};
  }
  ...
};
```

Then we can update the promise_type to provide an overload of return_value() that accepts an instance of inplace_construct.

```
template<typename T>
class task<T>::promise_type {
  std::suspend_point_handle<std::with_value<T>, std::with_exception> consumer_;
  std::continuation_handle continuation_;

  template<ConvertibleTo<T> U>
  void return_value(U&& value) {
    continuation_ = consumer_.set_value<T>((U&&)value);
  }

  template<typename... Args>
    requires Constructible<T, Args...>
  void return_value(in_place_construct<Args...> inplace) {
    continuation_ = consumer_.set_value_from<T>(inplace);
  }

  void unhandled_exception() {
    continuation_ = consumer_.set_exception(std::current_exception());
  }
```

```
  auto done() { return continuation_; }

  // ... rest of the definition omitted for brevity.
};
```

Then, finally, we can rewrite the `co_return` statement in the coroutine to return an instance of `in_place_construct` instead of constructing the return-value itself. e.g.

```
task<some_type> example() {
  int x = co_await something();
  co_return in_place_construct{x}; // Equivalent to 'return {x};'
}
```

We could also provide an additional in_place_construct_type<T> factory that would allow annotating the co_return statement with the return-type:

```
task<some_type> example() {
  int x = co_await something();
  co_return in_place_construct_type<some_type>{x};
}
```

The former might require that the type is implicitly constructible, whereas the latter would allow calling both implicit and explicit constructors.

Another variation of in_place_construct that accepts a callable and that constructs the return-value using the result of invoking a callable would also be required to achieve return-value optimisation for cases where the return value was obtained from a normal function call.
e.g.

```
some_type create(int i);

task<some_type> without_rvo() {
  int x = co_await something();
  co_return create(x);
}

task<some_type> with_rvo() {
  int x = co_await something();
  co_return in_place_construct_call{[&] { return create(x); }};
}
```

Another case to be considered is where the return value-expression is the result of a `co_await` expression. e.g.

```
task<some_type> other_function(int i);

task<some_type> example() {
  int x = co_await something();
  co_return co_await other_function(x);
}
```

It is not yet clear whether the same approach can be taken to achieve this kind of RVO for general co_await expressions. More investigation is required.

Note that the same approaches used for `co_return` could also be applied to `co_yield` statements. This would allow yielding values from a generator that are constructed in-place in the consumer's frame.

Language support for avoiding construction of temporaries

While the library solution to avoiding construction of temporaries in `co_return` statements is functional, it requires unnatural and verbose syntax for returning the result.

Ideally, the programmer could just write '`co_return T{args...};`' instead of having to write '`co_return in_place_construct_type<T>{args...};`'.

However, now that we have a library solution we can consider adding language syntax that is simply syntactic sugar for the library solution, in the same way that uniform initialization syntax can be syntactic sugar for a call to the `std::initializer_list` overload of a constructor.

We could add a rule for `co_return` and `co_yield` expressions that requires the compiler to check if the `promise.yield_value()`/`return_value()` methods are callable with an instance of the corresponding `std::in_place_construct_*` type and if so then invokes the method with an instance of that type rather than with the result of the expression.

User Syntax	Potential RVO Lowering (if the expression is valid)
`co_return T{args...};`	`promise.return_value(` `  std::in_place_construct_type<T>{args...});`
`co_return {args...};`	`promise.return_value(` `  std::in_place_construct{args...});`

`co_return some_function(args...);`	`promise.return_value(`
	`  std::in_place_construct_from{`
	`  [&](auto&&... args) {`
	`  return [&] {`
	`  return some_function(`
	`  (decltype(args)&&)args...);`
	`  };`
	`  }(args...)});`

Note that there are parallels between this approach and that of P0927R0 - "Towards A (Lazy) Forwarding Mechanism for C++". More investigation is required to determine whether the approaches can be combined.

Avoiding copies for nested calls

In the previous section we identified a challenge with trying to implement support for in-place construction of the return-value when the operand to `co_return` was itself a `co_await` expression.

This means that use-cases where an async coroutine is returning the result from another async coroutine can still involve an extra copy/move when returning the result. e.g

```
task<some_type> foo(int x);

task<some_type> bar() {
  int x = co_await something();
  co_return co_await foo(x);
}

task<void> baz() {
  some_type result = co_await bar();
}
```

In this case, while the foo() coroutine may be able to directly emplace the result of the `co_await foo(x)` expression without needing to make a copy, that result is still a temporary and we would need to move that temporary into the return-value of the `co_await bar()` expression.

The preferred solution would be to have compiler magic that somehow constructs the suspend-point handle for `co_await foo(x)` so that it's result is constructed in-place at the address of the 'result' variable, as this would be the most general solution. More investigation is required to determine if this will be feasible, however.

In the mean-time we can pursue creating a library solution by defining a task<T> type that implements the ability to hook up RVO for nested calls to other task<T>-returning coroutines that are called in the tail-position.

The idea would be to allow the programmer to write '`co_return foo(x);`' instead of '`co_return co_await foo(x);`' and have this pass the suspend-point handle of `bar()`'s caller into `foo()`'s task so that `foo()` can directly construct its result in the final location.

The solution to this would depend on two other extensions to the `co_return` statement, however.

The relaxation of the `co_return` restrictions as described in P1713R0 would be necessary to allow a generic implementation to support this syntax uniformly for both `task<void>` and `task<non-void>`.

This would also require extending the co_return statement to itself be a suspend-point so that the coroutine could suspend at the co_return expression until the nested coroutine had run to completion. This extension is described in P1745R0 in the section "Add support for co_return as a suspend-point".

In the mean-time, one workaround could be to use `co_yield` instead of `co_return`, which is already a suspend-point and, with the addition of the `set_done()` method described in P1662R0, can be made to behave as if it is a `co_return` statement by resuming with the '`goto done;`' continuation.

However, the compiler would not be able to make the same assumptions about control flow and dead-code with a `co_yield` expression as it could with a `co_return` statement.

Adding support for this incrementally

The ability to support return-value optimisations for coroutines is not something that can be added to C++20 at this stage. It has not been implemented and the impacts that this design would have on libraries is not yet fully understood.

If this is something we decide is important to add to the C++ language in the future then we can look at adding it incrementally on top of the version of coroutines that ships in C++20.

Adding support for async RVO later would introduce a second Awaitable concept to the language but this would be able to be used side-by-side with awaitable objects implemented using the C++20 Awaitable concept.

This needs to be viewed in combination with the changes proposed in [P1745R0] as the path and end-result will differ based on whether the changes in that paper are adopted for C++20.

Conclusion

This paper shows a potential path to incrementally add support for async RVO in a future release of C++ after C++20.

However, the viability of this approach depends on the adoption of changes in P1745R0 that split the `coroutine_handle` interface into separate SuspendPointHandle and ContinuationHandle concepts.

This paper recommends adoption of the changes in P1745R0 to enable this future evolution path for coroutines post-C++20.

Acknowledgements

Many thanks to Kirk Shoop for feedback and comments on draft versions of this paper.

Appendix - Examples

Example A: An async I/O operation that completes either with a value or an error.

```
using os_callback_t = void(std::error_code, size_t, void*);
void os_async_read(os_handle_t handle, void* buffer, size_t count,
                   os_callback_t* cb, void* userData);

struct async_read_op {
  os_handle_t handle;
  void* buffer;
  size_t count;
  coroutine_handle<> coro;

  template<typename Handle>
  auto operator co_await(Handle h) {
    coro = h;
    os_async_read(handle, buffer, count,
                  &async_read_op::on_complete, static_cast<void*>(this));
    return std::noop_coroutine();
  }

private:
  static void on_complete(std::error_code error, size_t bytesRead, void* userData) {
    auto* op = static_cast<async_read_op*>(userData);
    if (error) {
      // Operation failed - resume with exception
      auto continuation = op->coro.set_exception(
        std::make_exception_ptr(std::system_error{error}));
      continuation();
    } else {
      // Operation succeeded - resume with value
       auto continuation = op->coro.set_value<size_t>(bytesRead);
       continuation();
    }
  }
};
```

Example B: Implementation of task<T> for existing coroutines design and with support for RVO

```
template<typename T>
struct task {
  struct promise_type {
    std::coroutine_handle<> continuation_;
    std::variant<std::monostate, T,
                 std::exception_ptr> result_;

    ...

    template<typename U>
    void return_value(U&& value) {
      result_.emplace<1>((U&&)value);
    }

    void unhandled_exception() {
      result_.emplace<2>(
        std::current_exception());
    }

    auto final_suspend() {
      struct awaiter {
        bool await_ready() { return false; }
        auto await_suspend(
          coroutine_handle<promise_type> h) {
          return h.promise().continuation_;
        }
        void await_resume() {}
      };
      return awaiter{};
    }
  };

  struct awaiter {
    coroutine_handle<promise_type> coro_;
    bool await_ready() {
      return false;
    }

    auto await_suspend(coroutine_handle<> h) {
      coro_.promise().continuation_ = h;
      return coro_;
    }

    T await_resume() {
      auto& result = coro_.promise().result_;
      if (result.index() == 2) {
        std::rethrow_exception(
          std::get<2>(std::move(result)));
      }
      return std::get<1>(std::move(result));
    }
  };

  awaiter operator co_await() && {
    return awaiter{coro_};
  }

  ...

private:
  coroutine_handle<promise_type> coro_;
};
```

```
template<typename T>
struct task {
  struct promise_type {
    std::suspend_point_handle<
        std::with_set_value<T>,
        std::with_set_exception> sp_;
    continuation_handle cont_;

    ...

    template<typename U>
    void return_value(U&& value) {
      cont_ = sp_.set_value<T>((U&&)value);
    }

    void unhandled_exception() {
      cont_ = sp_.set_exception(
        std::current_exception());
    }

    auto done() {
      return cont_;
    }
  };

  template<typename Handle>
  auto operator co_await(Handle sp) && {
    coro_.promise().sp_ = sp;
    return coro_.resume();
  }

  ...

private:
  using handle_t = std::suspend_point_handle<
    std::with_promise<promise_type>,
    std::with_resume, std::with_destroy>;

  handle_t coro_;
};
```