Incremental Approach: Coroutine TS + Core Coroutines

Document Number: **P1362 R0** Date: 2018-11-15
Reply-to: Gor Nishanov (gorn@microsoft.com)
Audience: WG21, EWG

Abstract

Core Coroutine paper [P1063R1] seeks to address some of the limitations of Coroutines TS [N4775] that were mentioned in [P0973R0] and offers two ways to move forward:

• Delay the merge of Coroutines TS into the IS and wait until the exploration of the alternative design is completed (estimated completion of the exploration moves Coroutines out of C++20).

• Incremental: Merge the Coroutines TS and deliver solutions to the concerns mentioned as the solutions become ready for adoption.

This paper evaluates pros and cons of both options and **strongly** recommend pursuing the **incremental** approach and outlines a possible roadmap of what coroutine capabilities could be available in C++20 and which can come in later releases.

Contents

1 Overview

The R1 revision of the Core Coroutine paper is a significant and welcomed evolution over the R0 version. The user facing coroutine syntax, the layering of the coroutine machinery and the customization points are now much **closer** to the Coroutine TS design.

Core Coroutines provide more **elegant** and **efficient** support for the use case of unwrapping of `expected<T>` **than** what is possible with Coroutines TS today. For asynchronous cases, Core Coroutines **does not** yet allow authoring asynchronous coroutines with the same performance characteristics and does not yet support the same variety of use cases as the Coroutines TS.

Both designs share a similar layering approach:

Syntax Layer	User facing syntax of the coroutine
Library Layer	Library guided transformation imbuing the coroutine with high-level semantics (generator, async task, etc.)
Mechanism Layer	Internal coroutine mechanism of transforming a function into a state machine

The fundamental difference between Core Coroutines and the Coroutines TS lies in the library layer.

The Coroutines TS does not expose coroutine state machine object directly and uses compiler-based wrapping of the coroutine state machine guided by a set of customization points. Core Coroutines, on the other hand, provides a manifestation of the coroutine state machine as a function object that allows implementing most of the coroutine wrapping algorithm in the library and requires significantly fewer customization points[1].

In early design stages of the Coroutines TS (known back then as resumable function v2 [N4134]), exposing the coroutine state machine as a function object was our **first design choice** [GDRGOR2014]. We encountered several design and implementation challenges that led us to the decision to keep the Coroutine State Machine hidden until the design challenges are resolved, since we can achieve desired efficiency and usability goals for asynchronous and generator use cases immediately. We also anticipated that at some point in the future, the design and implementation challenges might be resolved and manifestation of the Coroutine State Machine as a function object can be added in a non-breaking fashion that would allow to cover more use cases and enable more flexible coroutine wrapping approach.

Therefore, the **incremental** path offered in the Core Coroutine paper is anticipated and fully aligned with Coroutine TS design and, in fact, very **welcomed**. We appreciate the flexibility of having a coroutine state machine manifested as a concrete object, and we would love to have it when the language design and implementation challenges are resolved, and deployment and usability experience is collected and found favorable.

On the other hand, the delay and wait approach that positions the Core Coroutines as the wholesale replacement of the Coroutines TS with a brand new, untested design, requires solving many implementation and language design challenges and it is uncertain whether the end result will be a satisfactory replacement.

[1] Though as shown in section 3, to solve the same set of use cases with the same efficiency as Coroutines TS, Core Coroutines might need to acquire a similar number of customization points as Coroutines TS does today.

To be able to better evaluate pros and cons of both approaches we will consider major design challenges that need to be addressed by the Core Coroutines proposal and explore possible solutions for small issues that needs resolution to reach performance of the Coroutines TS and to cover the same variety of use cases.

2 Design Challenges of explicit Coroutine State Object

The set of challenges described in this section need to be solved by any coroutine design that choses to expose manifestation of Coroutine State Object as a function object or a class of some other shape. That includes Core Coroutines, Resumable Lambdas [N4244], Resumable Expressions [P0114R0] and Named Class Coroutines [P1329R0] (described on page 8).

These challenges are:

- Sizeof challenge
- Tail call challenge
- Optimization challenge
- Usability challenge

Core Coroutines does not yet offer complete solutions to these challenges. The usability challenge is nearly solved, and we expect to see the details in the next revision. The remaining three challenges do not have an obvious solution and we do not know the timeline when the solutions will be available and what will be the language impact, implementation cost and other properties of the proposed solutions.

2.1 Sizeof challenge

"… Coroutine Uncertainty Principle: A coroutine can either be fast or you can know its size." — Heisenberg?

Background: Independently of each other, major compiler vendors, specifically, GCC compiler engineers, Google compiler engineers working on Clang and MSVC compiler engineers have reached the same conclusion. Namely, to achieve efficient transformation of a function into a state machine, this transformation must be done after optimization passes have simplified the body of the function.

Given the desire to have efficient coroutines, it is challenging for the frontend of the compiler to reliably estimate the size of the coroutine during semantic analysis. This conclusion is also shared by EDG's Daveed Vandevoorde.

It also reflects the current thinking of the Core Coroutine authors. While the initial design choice for Core Coroutines was to expose the "**sizeof**" of the coroutine state object in a type system as indicated by r0 and r1 revisions of the Core Coroutine paper, it was stated during the presentation to EWG of Core Coroutines r1 in San Diego, that it is no longer the case.

Currently, Core Coroutine design team is exploring **introducing no-sizeof** types into the C++ type system. The detailed design is not available yet, but it was stated that it is likely to include such features as, being able to place no-sizeof types on the stack, embed them as a last field of a struct (which in turn makes the enclosing struct to be no-sizeof type as well), etc.

This is an open design question that needs to be solved by Core Coroutines proposal. We don't know when an acceptable solution will be found and whether the committee will accept a complexity that such solution might

entail. Coroutine TS addresses this challenge by not exposing coroutine state machine object directly and allowing the user to control the placement/allocation of the coroutine state via customization points (section 4.3).

2.2 Tail call challenge

"… To go any lower-level than that, you'd need a miniature soldering iron and a very, very steady hand." — Andrei Alexandrescu

As the Core Coroutine paper correctly points out, a coroutine transformation can easily turn an innocuously looking user code such as a loop into an unbounded recursion. Without provisions for symmetric coroutine-to-coroutine transfer, library facilities implementing coroutine high-level semantic must provide workarounds that include (depending on the application), scheduler loops, trampolines and even atomic operations<reference>.

The R1 revision of the Core Coroutine paper suggests a limited tail call facility that is sufficient for handling the use case of unwrapping of `expected<T>`, but it is inadequate to implement unrestricted coroutine-to-coroutine symmetric transfer available in the Coroutines TS. The latter is an important requirement for efficient handling of asynchronous and recursive generator use cases.

As we understand, one of the possible solutions to bridge the gap would be to add tail-calls into the type system so that it becomes part of the function signature (and thus the ABI). That would make it possible to perform tail calls across ABI boundaries and tail calls for indirect function calls.

Tail call handling is an open design question for Core Coroutines.

Coroutines TS, as usual, addresses this challenge by not exposing the coroutine machine state object and instead relying on compiler-based wrapping to synthesize appropriate calling convention and fuse suspension of the coroutine with resumption of another into a tail call.

2.3 Usability challenge

"What would happen if an irresistible force met an immovable object?" — Irresistible force paradox

The coroutine state machine object is by necessity an immovable object: it cannot be moved and it cannot be copied. The problem of correctly copying or moving the coroutine state is similar to the problem of synthesizing the **correct** move or copy constructor for an arbitrary user class. This is recognized by the Core Coroutine proposal authors and Core Coroutine state machine object has its copy and move constructor and assignment deleted.

Coroutine state machine object is not useful by itself and requires wrapping into semantically meaningful class. Usually, a type-erased wrapper that involves heap allocation and type erasure **or** an embedded wrapper which wraps the immutable coroutine objects into semantically meaningful immovable class.

The following is an illustration of user facing syntax for both.

Type-erased coroutine	Embedded coroutine (expected to be simpler in r2)

`auto Traverse(BstNode<int> *node)[node][->]generator<string> {`
`  if ( node == nullptr )`
`  return;`
 	
`  [<-] Traverse ( node -> left );`
`  [<-] std::yield ( node -> value );`
`  [<-] Traverse ( node -> right );`
`}`
	`template <typename Range>`
	`auto traverser(const Range& range) {`
	`  using Sg = stack_generator<decltype(*begin(range)), void>;`
	`  return Sg([&] {`
	`  return [&] [->] Sg {`
	`  for ( auto & element : range )`
	`  [<-] std :: yield ( element );`
	`  });`
	`  });`
	`}`

While we expect that r2 of Core Coroutines paper will bring embedded coroutine syntax closer to that of the type-erased coroutines, there are still usability challenges associated with the approach of embedding of the concrete coroutine state machine into the return type.

Type-erased coroutines	Embedded coroutines
✓ Can be forward declared	✘ Cannot be forward declared
✓ Can be called recursively	✘ Cannot be called recursively
✓ Can be put on ABI boundary	✘ Cannot be put on ABI boundary[2]
✓ Does not require solution to sizeof challenge	✘ Requires solution to a sizeof challenge
✓ Definition can reside in the implementation file	✘ Definition must reside in the header / module interface file
✓ Can be virtual	✘ Cannot be virtual
✘ BUT!!! Requires heap allocation & indirect calls	✓ Does not require heap allocation

When working on the early design of the Coroutines TS in 2014, our observation was that due to restrictions on usability of embedded coroutines, it is very likely that a large number (if not most) of the coroutines will follow the type-erased pattern and therefore we needed to design for efficient type-erased coroutines. Which brings us to the next challenge.

2.4 Optimization challenge

Given the expectation that a large number of coroutines would follow type-erased patterns, the Coroutines TS design (since 2014) includes provisions to undo type-erasure and heap allocation for the coroutines. It was enshrined in the core wording for the Coroutines TS and implemented in two compilers (Clang and MSVC). A similar optimization is expected to appear in GCC compiler once the implementation of coroutines lands. Conditions for when the optimization can apply were recently summarized in the Halo paper [P0981R0].

The Coroutines TS is able to implement Halo and other optimization in a straightforward manner as the algorithm for wrapping the coroutines is performed by the compiler itself and a coroutine specific markup is inserted in the intermediate representation. The mark up provides information to the optimizer that allows to undo the type-erasure and heap allocation when appropriate.

If the wrapping is performed by the library, the challenge is how to remove indirect calls and heap allocations in the same variety of scenarios possible with the Coroutines TS. This is an open design question and no solution is offered by the Core Coroutine paper.

[2] otherwise extremely fragile, as any change to implementation immediately leaks through an interface.

2.5 Bonus Challenge

"when does a [snip] system stop existing as a superposition of states and become one or the other?" — Erwin Schrödinger

Finally, a coroutine design that choses to expose a coroutine state machine needs to decide on its exact shape. The challenge is to decide which one is the right one. The design space is wide and there are many participants.

- Is it a Resumable Lambda? [N4244]
- Is it a state machine from Resumable Expressions? [P0114R0]
- Is it a state machine from Core Coroutine [P1063R0], (more powerful, but more complicated than r1)?
- Is it a state machine from Core Coroutine [P1063R1], (less powerful, but simpler)?
- Is it a state machine factory from the Unified Coroutines [D1342R2] paper by Facebook ?
- Is it a state machine from the Named Class Coroutines proposal (to appear in pre-Kona mailing)?

Did we explore the concrete coroutine object design space enough? Did we have implementations, and did we collect the deployment experience? Do we know performance trade-offs of one over the other? Are they implementable? Do we restrict optimization opportunities if we fix the shape of the coroutine to a particular class?

The Coroutines TS sidesteps these hard questions by saying, we will not expose a state machine object for now and leave the decision about the concrete state machine open for future evolution.

3 Solving minor issues with Core Coroutines

In the previous section we covered a set of difficult problems that Core Coroutines need a solution to be implementable and to match the efficiency of Coroutines TS. In this section, we look at small issues with Core Coroutines that are straightforward to fix, but it is likely that fixing them will bring number of customization points much closer to what we have in Coroutines TS and therefore will reduce even further the difference between Coroutines TS and Core Coroutines.

3.1 Baseline (5:5)

Let's start with the customization points which are common to both Core Coroutines and Coroutines TS.

Coroutines TS	Core Coroutines
promise_type	shared_state_type
await_suspend	coroutine_suspend
await_resume	coroutine_resume
return_void()	coroutine_return()
return_value(T)	coroutine_return(T)

Historical note: Early revision of Coroutines TS, for example, [N4134] also used arity of the return statement customization point as Core Coroutines does now. Subsequent revisions split it up into distinct `return_void` and `return_value` to avoid surprising behavior if library defined the customization point with default arguments that would enable both (`return;` and `return 42;` statement within the same coroutine) and for other reasons.

3.2 Final Suspend (6:6)

In Coroutines TS, for asynchronous tasks and recursive generators, `final_suspend` point is used by the library defining coroutine semantics to tail resume the coroutine that is awaiting on the one that has just completed.

Let's explore possible alternatives to having a final suspend.

Can we tail resume the awaiting coroutine from `coroutine_return` customization point?

- Not if a `return` is in a catch block. ABI restrictions prevent tail calls from the catch blocks.
- If `coroutine_return` resumes the awaiting coroutine, locals in the returning coroutines are not destroyed yet. This means that RAII destructors that might have released the mutex, signaled the event have not been fired yet. This will result in deadlocks and other surprising behavior.

Recognizing aforementioned problems Coroutines TS introduces a final_suspend point. A point in the coroutine where all of the locals are destroyed and where unrestricted tail calls are possible when needed. It is very likely that Core Coroutines will acquire an equivalent of `final_suspend` as the authors elaborate further on their design.

3.3 Unhandled Exception (7:7)

Coroutines TS wraps the user authored body of the coroutine in a try-catch and forwards any unhandled exceptions from the user authored body to the `unhandled_exception` customization point.

Core Coroutine paper makes a conjecture that `unhandled_exception` customization point is not needed and can be instead handled by the code resuming the coroutine. Here is a rough sketch of how it might look like.

Coroutines TS	**Core Coroutines**

`void scheduler() {`
`  while (auto h = q.pop_front())`
`  h.resume()`
`}`
	`void scheduler() {`
	`  while (auto h = q.pop_front())`
	`  try { h.resume();}`
	`  catch(…) {`
	`  auto e = std::current_exception();`
	`  // figure out who was awaiting on coroutine h`
	`  // inject exception e into the awaiting coroutine`
	`  q.push_front(<awaiting-coroutine>)`
	`  }`
	`}`

There are at least four problems with that solution (#2 being the killer, others are incidental).

1) To decide what to do a scheduler need to be intimately aware with a concrete coroutine type being resumed and how it connects to its awaiter (shorthand for awaiting coroutine here).
2) Even if #1 is solved, in the presence of tail calls, we don't know whether an exception that escaped the coroutine is from the coroutine we resumed or from some other coroutine it tailed called into, thus, leaving us completely helpless without any possibility of figuring out whom to give an exception and whom to resume.
3) There is also a minor performance implication - since we now must rely on the queue and a scheduler to resume the awaiting coroutine, whereas in Coroutines TS, awaiting coroutine is tail resumed from the final suspend point after capturing the exception in `unhandled_exception` customization point.
4) There is also a potential code bloat implication as now any code that need to resume the coroutine need to go through a similarly structured try-catch and resume dance.

Our conclusion is that Core Coroutine proposal would need to wrap the user-authored body in `try-catch` and forward any unhandled exception to an equivalent of the `unhandled_exception` customization point. It can also make it optional[3] as the wrapping is only important for asynchronous scenarios.

3.4 Await Ready (8:8)

Both Coroutines TS and Core Coroutines require that prior to invoking `await_suspend`/`coroutine_suspend` customization point currently executing coroutine is considered suspended[4] even though it is still running the code in the customization point.

This makes it legal for the code inside of `await_suspend`/`coroutine_suspend` to resume the coroutine (either in the same thread) or concurrently from a different thread. To achieve that, before calling `await_suspend/coroutine_suspend` a compiler must prepare the coroutine for suspension, the resulting code might looks like:

With await_ready	Without await_ready

`if (await_ready()) goto SuspendBypass;`
`  <spill registers into the coroutine frame>`
`  <store current coroutine suspend point>`
`  <store ABI required data, for example EH index on win-i386>`
`  call await_suspend/coroutine_suspend`
`  <jmp to the epilogue of the function>`
`ResumeLabel:`
`  <restore spilled registers>`
`  <restore ABI required data>`
`SuspendBypass:`
`  call await_resume/coroutine_resume`
	`  <spill registers into the coroutine frame>`
	`  <store current coroutine suspend point>`
	`  <store ABI required data, for example EH index on win-i386>`
	`  call await_suspend/coroutine_suspend`
	`  <jmp to the epilogue of the function>`
	`ResumeLabel:`
	`  <restore spilled registers>`
	`  <restore ABI required data>`
	
	`  call await_resume/coroutine_resume`

As seen in the example above, if an expression being awaited is ready, it allows us to bypass a lot of unnecessary operations. It is not clear that sufficiently clever optimizer can transform the code on the right into code on the left if it can analyze the body of `await_suspend`/`coroutine_suspend` and decide that such bypass is needed and whether the transformation is safe[5].

When designing Coroutines TS, whenever confronted with a choice of simpler customization points vs performance, we were choosing performance. Due to layering of Coroutines TS, customization points are generally invisible to vast majority of the coroutine users, therefore having another one, if it allowed us to produce more performant code is an appropriate choice.

---

[3] Similar simplification can be applied to Coroutines TS in C++20 timeframe.

[4] This makes it safe to invoke an operation that initiates asynchronous activity from await_suspend and give it a callable that will resume the coroutine when operation is completed. The resumption can happen concurrently on the other thread while the current thread is still in the middle of execution of await_suspend. Without this property, coroutines cannot compete with callbacks when it comes to consuming asynchronous APIs, since they would need additional synchronization that is not needed in the case of callbacks.

[5] To be safe, compiler needs to recognize and understand any OS or library function that are behind ABI boundary that are called from `await_suspend/coroutine_suspend`, as any of the functions that compiler cannot see can potentially resume the coroutine and make the transformation invalid.

We believe to match the performance of Coroutines TS for the same use cases, Core Coroutines would require addition of an equivalent of `await_ready`.

3.5 coroutine_traits/get_return_object (10:10)

Core Coroutines proposal requires two intrusive change to an existing type to be able to author coroutines returning that type.

1. Add a nested `shared_state_type` class to any user defined class that maybe used as a return value of the coroutine.
2. Add a constructor to that type taking a coroutine state object as an argument.

One of the design goals of Coroutines TS is incremental and seamless integration into existing codebases. C++ community is diverse. Some organizations have codebases where you can easily modify any line of the source code that goes into building your products, others, have code ownership restrictions that can make changes in libraries not owned by your team more challenging to make.

Coroutines TS recognizes the diversity of C++ community and offers a non-intrusive way of introducing coroutines into the code base.

Just like Core Coroutines proposal needs to find the `shared_state_type` to understand the meaning of a particular coroutine, Coroutines TS needs to find a `promise_type` that serves the same purpose.

Coroutines TS uses a specialization of `coroutine_traits<R,T1,T2,…>::promise_type` to find it, where `R,T1,…` are the return and argument types[6] of a coroutine in question. Predefined primary template of `coroutine_traits` conveniently looks for the nested type in the return type `R` of the coroutine, which results in identical behavior to the Core Coroutines if no specialization was provided and a `promise_type` happens to be intrusively added to as a nested type to type `R`.

To construct the return object of the coroutine, Coroutines TS uses `get_return_object` customization point. To summarize:

Coroutines TS	Core Coroutines
specialize coroutine_traits	(intrusively) add a nested shared_state_type to the return type
get_return_object	(intrusively) add a coroutine aware constructor to the return type

We believe that to serve the same diverse C++ community as Coroutine TS does, Core Coroutines would need to acquire an equivalent of `coroutine_traits` and `get_return_object` customization points.

3.6 operator co_await (11:11)

Early iterations of Coroutines TS have used a similar approach with respect to `await_suspend` and `await_resume` to what Core Coroutine proposal does with `coroutine_suspend/coroutine_resume`. Namely they were either member or free functions found via ADL that described how to await on a particular

[6] Coroutine TS uses arguments types in addition to the return type when specializing the `coroutine_traits` to handle a variety of use cases, from handling tricky coding conventions where the coroutine object is returned as an out parameter, to allowing using the same coroutine vocabulary type while augmenting the behavior based on the allocator, executor, interrupted token, actor passed in as one of the arguments to the coroutine.

expressions without intrusive changes to the UDT representing the result of the operand of the await/[<-] operator.

Soon after deployment in 2014, a pattern has emerged of bundling `await_ready/suspend/resume` functions together in a struct (colloquially called awaiter). The benefit of this approach was that it allowed to keep per suspend point state controlled by the designer of an awaiter. It was used to either keep per operation state that otherwise would need to be heap allocated or to keep other state related to the operation and make it available to `await_ready/suspend/resume` as needed.

It was not possible to add full powered awaiter to a UDT type in a non-intrusive fashion using free functions `await_ready/suspend/resume`. To address this deficiency, 2015 revision of the Coroutines TS has removed ADL discoverable free functions `await_ready/suspend/resume` and introduced `operator co_await` that provided a non-intrusive mapping of an arbitrary UDT to an awaiter.

The end-to-end story of Core Coroutine library has not been investigated in detail yet, and we expect that the need for something like `operator co_await` would be discovered in Core Coroutines with deployment experience to solve problems in asynchronous programming domains.

3.7 await_transform (12:12)

Coroutines TS partitions customization points into two concepts, one is the *Coroutine Promise* that defines the semantic of the coroutine and another is *Awaitable* that defines how a particular UDT type needs to be processed within a context of the coroutine when awaited upon.

The separation is based on the observation that coroutine types and awaitables could be developed independently of each other by different library vendors. In a product, one could use N coroutine types that describe how coroutine behaves and M awaitables describing how a particular asynchronous API should be consumed. Users can freely mix and match coroutines and awaitables as needed.

Sometimes, an author of the coroutine type, may want to alter how an awaitable produced by some other library behaves in the coroutine of that particular type. For example, a coroutine that is cancellable may want to instrument every await point in the coroutine with a cancel check that will inject a cancellation before and after an await is performed. This capability is enabled by optional `await_transform` customization point that give the coroutine type designer the ability to transform an incoming awaitable and augment it with extra functionality as needed[7].

Core Coroutine proposal does not separate their customization points into separate categories and every `coroutine_suspend/coroutine_resume` depends both on the promise and the awaitable which at the moment would require providing implementation for N * M customization points, whereas Coroutine TS will get by with just N + M.

The end-to-end story of Core Coroutine library has not been investigated in detail yet, and we expect that a solution will be found to handle similar set of use cases as Coroutines TS, and it may require adding an equivalent of `await_transform` customization point.

[7] Besides cancellation, other known uses of the `await_transform` are instrumenting coroutines with tracing of suspension and resumptions of the coroutines, altering an execution context of where coroutine resumes, putting restrictions on what awaitables can be handled by a coroutine of a particular type and more.

4 Any other customization points?

In the previous section, while working to address performance and usability issues of Core Coroutine proposal we ended up discussing most of the customization points available in the Coroutines TS. In this section we will cover customization points that has not been covered so far.

4.1 yield-expressions

Having a dedicated yield-expressions in addition to await-expressions follows from the following conceptual framework.

Produces a … (how) …	**Synchronously**	**Asynchronously**
Single value	Plain old function	Task
Sequence of values	Generator	Async Range

The await-expressions implies getting a value (potentially asynchronously) into a coroutine. The yield-expression implies getting a value out of the coroutine (possibly asynchronously). Combination of two covers the entire design space in the table above.

It is technically possible to eliminate yield-expressions from the language and express generator and async-range purely in terms of await-expression using a helper tag type, for example:

Coroutines TS	**Coroutines TS (without yield-expressions)**

`generator<int> infinite_sequence(int start) {`
`  for (;;)`
`  co_yield start++;`
`}`
	`generator<int> infinite_sequence(int start) {`
	`  for (;;)`
	`  co_await std::yield(start++);`
	`}`

Removing yield-expression from the Coroutines TS will save one customization point (`yield_value`) at the cost of readability and removing conceptual symmetry between await (getting something into the coroutine) and yield (getting something out).

Another consideration that reinforced the decision to keep a dedicated syntax for yield is to keep the door open to enable return type deduction for coroutines[8]. In asynchronous functions, returning a conceptual Task<T>, type T would be deduced from the return statements, like in a regular function, however, in a conceptual Generator<T> and AsyncRange<T>, deduction of T would be performed from operands of yield-expressions.

4.2 Initial suspend

In section 3.2 we discussed final suspend point that can be thought of as an implicit suspend point occurring on the closing curly brace of the *function-body* of the coroutine. Initial suspend represents its symmetric counterpart that occurs on the open curly brace of the *function-body* of the coroutine.

[8] A version of coroutine return type deduction is available in MSVC and EDG compiles since 2015 and was part of the design since 2014. It is not part of the Coroutines TS at the moment as there are some design choices related to type deduction we would like to keep open for now.

Symmetry between initial and final suspend

```
task<int> async_sum(channel& s) { // <-- (implicit) co_await p.initial_suspend();
  int sum = 0;
  for (;;)
    sum += co_await s.async_read<int>();
  co_return sum;
} // <-------------------------------- (implicit) co_await p.final_suspend();
```

While it is technically possible to roll the functionality of initial_suspend into get_return_object(), we found the teachability and symmetry aspects of `initial_suspend` beneficial enough to keep it as a part of the Coroutines TS.

If we were to focus on minimizing number of customization points, it is possible to refactor the functionality currently performed by the `initial_suspend` into `get_return_object`. It will be a breaking change to the existing users of the TS, but it is unlikely there will be a functionality loss because of this change.

4.3 Coroutine State Allocation Control

We discussed in section 2 challenges related to the design choice of exposing a manifestation of a coroutine state machine as a function object so that wrapping of that state machine into a usable type could be done mostly with the library code, whereas Coroutines TS keeps the object hidden and perform the wrapping using compiler executed algorithm controlled by the customization points.

To control placement of the coroutine state, the coroutine designer can chose to provide `operator new` and `operator delete` in the `promise_type` of the coroutine. If provided, the coroutine will acquire storage for its state (if needed[9]) using provided operators `new` and `delete`. The designers of the coroutine type have freedom in how the control can be exposed to the users of the coroutine, one alternative, for example, is to use the approach similar to that of the standard library types that take an optional allocator argument indicated by the presence of `allocator_tag_t`.

Explicit allocation vs Implicit allocation

```
task<int> async_sum(channel& s);                          // uses default allocator for coroutine state

task<int> async_sum(allocator_tag_t, Alloc a, channel& s); // uses allocator a
```

To support the environments where exceptions are prohibited or unavailable, Coroutines TS support `nothrow` versions of `operator new` that are used in environments when out of memory is reported as returning `nullptr` from the operator new. This is controlled by the presence of `get_return_object_on_allocation_failure` customization point. If present, coroutine wrapping machinery adds a check after calling `operator new` if it returned `nullptr`, the constructor of the coroutine is aborted and the result of the invocation of the coroutine is produced by the invocation of `get_return_object_on_allocation_failure` customization point.

These three customization points are not needed if challenges explained in Section 2 are adequately resolved and library-based wrapping can match the efficiency and use cases of compiler based coroutine wrapping.

[9] Compiler is free to elide allocation of the state under the conditions described in Halo paper [P0981R0].

4.4 Are coroutines expert only feature?

Exploration of all customization points in Coroutines TS and Core Coroutine proposal may leave in impression that coroutines are incredible hard and expert only feature.

Luckily only a few (say, a thousand) of C++ developers in the entire world would have to know or use any of the coroutine customization points. Overwhelming majority (two millions?) of coroutines users would never have to learn, write or see a coroutine customization point.

Coroutine complexity is managed by layering. Coroutines are only consumed via high-level syntax, only those defining new coroutine types or awaitables need to go below the syntax layer.

Who	What
Everybody (millions)	Uses coroutines via high level syntax powered by coroutine types and awaitables defined by the standard library, boost and other high-quality libraries.
Power user (10,000)	Aware of *Awaitable* concept. Defines new awaitables to customize await for their environment using existing coroutine types
Expert (1,000)	Aware of *Awaitable* and *Coroutine Promise* concepts. Defines new coroutine types
Cream of the crop (200)	Defines metafunctions, utilities and adapters that can help to compose awaitables, write utility coroutine adapters, etc.

For typical uses, to define an awaitable, a programmer needs to be aware of 3 customization points:

```
struct execute_on {   // usage: co_await execute_on{e};
  executor e;
  bool await_ready() { return false; }        // 1
  template <typename F>
  void await_suspend(F f) { e.execute(f); }   // 2
  void await_resume(){}                       // 3
};
```

To define a simple coroutine type, it is sufficient to know only 5. Here is an example of the `promise_type` that can enable authoring of coroutine returning `std::future<R>`:

```
struct promise_type {
  promise<R> p;
  future<T> get_return_object() { return p.get_future(); } // 1
  suspend_never initial_suspend() const { return {}; }     // 2
  suspend_never final_suspend() const { return {}; }       // 3
  template <class U> void return_value(U&& value) {        // 4
    p.set_value(std::forward<U>(value));
  }
  void unhandled_exception() { p.set_exception(std::current_exception()); } // 5
};
```

5 Evaluating alternatives

Now that we acquired background on design issues and understand coroutine customization points, we are equipped to evaluate both paths forward offered in the Core Coroutine paper.

To do an estimate of when the Core Coroutine proposal can reach maturity and deployment experience comparable to that of the Coroutines TS we will look at the timeline of the evolution of the Coroutines TS:

- 2012 – 2013: Early design, simple implementation, handles only a few of the use cases
- 2014: Complete design handling all use cases. Implementation shipped in an official release of the MSVC. We start accumulating feedback from C++ developers across the world.
- 2015: Feature freeze. No substantial changes afterwards. Implemented in EDG frontend. Production use in MSVC compiler, Prototype in Clang compiler.
- 2016: Production deployment on Linux using Clang compiler (using a public fork)
- 2017: Coroutines are in a Clang 5.0. TS is published.
- 2018: GCC implementation starts, first coroutine library types proposed for standardization

By the time we are in Kona (2019), the Coroutines TS would have accumulated:

- 5 years of feedback of C++ developers of using the Coroutines TS (4 years in its current form)
- 4 years of production deployment on Windows in business-critical software
- 3 years of production deployment on Linux in business-critical software
- Implementation experience in 4 major compilers: MSVC, Clang, EDG and GCC

Assuming the same level of scrutiny is applied to the Core Coroutines proposal as it was the case with the Coroutines TS, we expect that the Core Coroutine proposal will reach maturity in the years 2022-2023 and will become eligible to be merged in C++26. The following table helps us to make the evaluation:

Year	Incremental Approach	Delay and Wait approach
C++20	Coroutine TS is merged. Coroutines can be used with boost::asio and other async libraries.	
C++23	Executor, Networking TS and Concurrency TS are merged and take advantage of integration with Coroutines. Ready parts from Core Coroutines are integrated. Possibly: * capture syntax from Core Coroutines * efficient handling of expected<T>	
C++26	More ready parts from Core Coroutines are integrated. Possibly: *concrete state machine object exposed”	Core Coroutines are merged. They have ~5 fewer customization points

6 Conclusion

“It is fundamental for any evolutionary strategy that the language is viable at every point in time; we cannot ignore serious challenges for a longish period of time, waiting for a perfect solution” – Bjarne Stroustrup

Benefits of the incremental approach are numerous:

- We benefit from 6 years of feedback based on wide deployment of coroutines and have two C++ release cycles to improve the coroutines if needed.
- We have integration of coroutines with library facilities introduced in C++23 and C++26.
- We have served C++ developers for 6 years addressing a major challenge of asynchronous programming and other uses cases.

Benefits of the “wait for much nicer coroutines to come in C++26” are few and the outcome of having nicer coroutines in C++26 is far from certain given the design challenges facing the Core Coroutines and given that it needs to be compared to the solution that is the result of an incremental approach that is based on a mature Coroutine TS and has benefited from two cycles of feedback and improvements in C++23 and C++26 and adopted parts of Core Coroutine proposal incrementally.

7 Acknowledgements

Many thanks to those who reviewed the drafts of this paper and provided valuable feedback, among them: Bjarne Stroustrup, Daveed Vandevoorde, Herb Sutter, Gabriel Dos Reis, Iain Sandoe, James Dennett, Mathias Stearn, Mihail Mihaylov, Lewis Baker, Peter Dimov and Thomas Köppe.

8 Bibliography

[N4134] G. Nishanov, J. Radigan. “Resumable Functions v.2” (WG21 paper, 2014-10-10).

[N4244] Christopher Kohlhoff. “Resumable Lambdas” (WG21 paper, 2014-10-13).

[N4775] “Working Draft, C++ Extensions for Coroutines” (WG21 paper, 2018-10-07).

[P0114R0] Christopher Kohlhoff. “Resumable Expressions” (WG21 paper, 2015-09-25).

[P0973R0] G. Romer, J. Dennett. “Coroutines TS Use Cases and Design Issues” (WG21 paper, 2018-03-23).

[P0975R0] G. Nishanov. “Impact of coroutines on current and upcoming library facilities” (WG21 paper, 2018-03-10).

[P0976R0] Bjarne Stroustrup. “The Evils of Paradigms Or Beware of one-solution-fits-all thinking” (WG21 paper, 2018-03-06).

[P0978R0] G. Nishanov. “A Response to "P0973r0: Coroutines TS Use Cases and Design Issues"” (WG21 paper, 2018-03-31).

[P0981R0] Richard Smith, Gor Nishanov. “Halo: coroutine Heap Allocation eLision Optimization” (WG21 paper, 2018-03-18).

[P1063R0] Geoff Romer, James Dennett, Chandler Carruth. “Core Coroutines” (WG21 paper, 2018-05-06).

[P1063R1] Geoff Romer, James Dennett, Chandler Carruth. “Core Coroutines” (WG21 paper, 2018-10-05).

[P1241R0] Lee Howes, Eric Niebler, Lewis Baker. “In support of merging coroutines into C++20” (WG21 paper, 2018-10-08).

[GDRGOR2014] Gabriel Dos Reis and Gor Nishanov. “Lambda*: Brainstorming and whiteboarding” in April of 2014.

[P1329R0] Mihail Mihaylov, Vassil Vassilev. “On the Coroutines TS” (page 8, WG21 paper pre-publication draft, 2018-11-02).

[D1342R2] Lewis Baker. “Unified Coroutines” (WG21 paper pre-publication draft, 2018-11-08).

[P0323R3] V. Botet, JF Bastien. “Utility class to represent expected object” (WG21 paper, 2017-10-15). (Current design paper for `expected<T,E>`.)

[P0380R1] G. Dos Reis, J. D. Garcia, J. Lakos, A. Meredith, N. Myers, B. Stroustrup. “A Contract Design” (WG21 paper, 2016-07-11).

[P0779R0] N. Douglas. “Proposing `operator try()`” (WG21 paper, 2017-10-15).

[Stroustrup 1994] B. Stroustrup. *The Design and Evolution of C++* (Addison-Wesley, 1994).