Coroutines: Language and Implementation Impact

Document Number: **P1492 R0**

Date: 2019-02-19

Audience: WG21, EWG

Authors: Richard Smith, Daveed Vandevoorde, Geoffrey Romer, Gor Nishanov, Nathan Sidwell, Iain Sandoe, Lewis Baker

Abstract

The EWG chair has requested a paper from the authors of various coroutine proposals and compiler experts to evaluate language and implementation impacts of several design decisions made by coroutine proposals under consideration.

Contents

1 Overview

A coroutine is a generalization of a function: regular functions always start at the beginning and exit at the end, whereas coroutines can also suspend execution to be resumed later at the point where they were left off. Although this concept is one of the older devices of compiler engineering, the C family of languages is, at present, without a built-in representation for the generalisation.

There are differences in currently proposed solutions related to how the resumable concept should be represented (and at what level of granularity)[1]. In some cases, the overt requirements of a given approach lead to derived requirements on changes to the underlying language or its implementation in a realisable toolchain.

For example, when a regular function executes, its required state is captured in the activation record; this record is [currently] private to the implementation (which also controls its creation and destruction), and is also permitted to have a different size on each invocation of the function. Nevertheless, since they are strictly nested, it is easy to reason about function activation record lifetimes.

A coroutine requires the equivalent of such an activation record[2], although the coroutine variant must support persistence across several suspension/resumption points and will generally escape from the context in which it is first created. The latter properties lead to the desire to model the coroutine activation record (or parts of it) as an object or closure of some form, so that its lifetime may be observed and reasoned about in a familiar manner.

However, removing the privacy (to the implementation) of the activation record and exposing all (or part) of it leads to at least two of the main topics to be discussed in this paper:

- The implementations of all modern optimizing compilers known to us strongly assume the privacy of the activation record.
- If part of the activation frame is exposed as a user-visible complete type, then there is a strongly embedded expectation of the language that its size and layout are known very early in the compilation pipeline (early enough to be able to produce sizeof and alignof values).

A common implementation strategy for a coroutine is a compiler-based transformation of a function into a state machine (expanding keywords that cause suspension into the control flow and state preservation required).

- This paper examines several approaches to how to do this transformation and how (or if) the state machine can be exposed in the language.
- Differing schemes also affect perceived usability (e.g. complexity or size of the user-facing portions of the designs).

Thus, we will look at implementation, usability and language impacts of various approaches.

There are also minor issues like the exact spelling of keywords which, while they might consume vast amounts of time in debate, are not central to the feasibility of any of the proposed solutions. Those issues are not discussed here.

Current set of proposals:

[1] The expression of whether what the compiler implements should be more (or less) hidden appears as “early split” and “late split” in the following discussions.

[2] This variant of the activation record is termed the “coroutine state object” in the following sections.

- Coroutines TS (current working draft: N4775)
- Core Coroutines (P1063R2)
- Symmetric Coroutines (P1430R0)
- Unified Proposal (P1342R0)
- Coroutine TS Plus (Coroutines TS + accrued improved by the time other proposals mature: P1362R0)

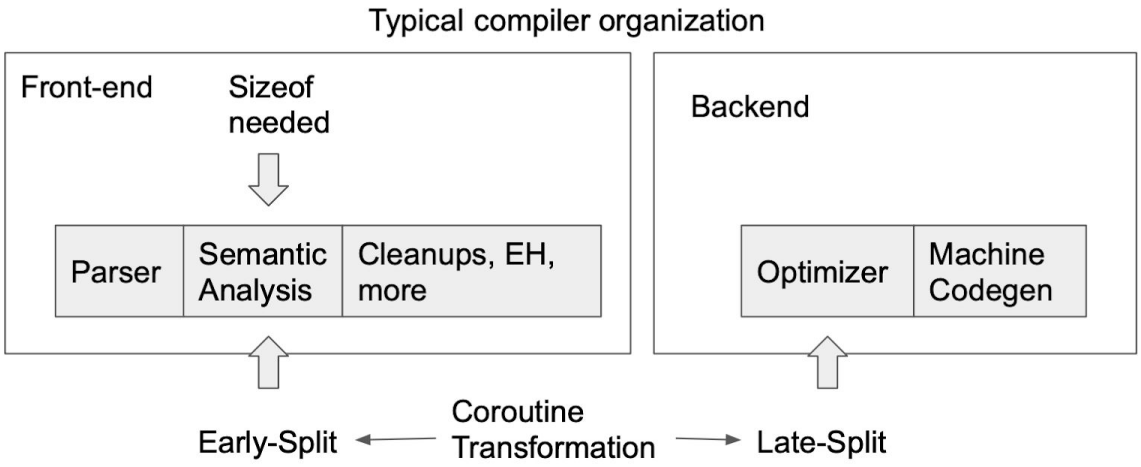
Objective:

To identify, as far as possible, the pre-existing barriers to key features of the various proposals.

2 Trade-offs at a glance

We're currently facing the following set of proposals:

- Coroutines TS today: Hidden State Machine + Late-split
- Core Coroutines: First Class Object + Late-Split/late-sized types
- Symmetric Coroutines: First Class Object + Early-Split / Late-Split (not decided yet)
- Unified Proposal (P1342r0, Lewis):
  - Option A: create_dynamic() only (Hidden State Machine)
  - Option B: create_dynamic() + create_static() returning early-sized types
  - Option C: create_dynamic() + create_static() returning deferred-layout types
- Coroutine TS Plus: Coroutine TS in C++20 and some type of First Class State Machine later based on implementation and deployment experience



Before we dive into the details, let's take a bird's-eye view of the trade-offs and challenges of various approaches, where the key issues (to be expanded upon later) are identified as:

- user-facing surface (size and complexity / number of library customization points)
- when or how the coroutine's state might be saved on the stack versus the heap
- effect on current compiler optimization schemes
- effect on current compiler and toolchain implementation structure
- effect on underlying language assumptions (primarily in the type system and the ability to determine things like sizeof and alignof for the representation of the coroutine state)

The following table attempts to summarize the various proposals and strategies that have been discussed:

Table: Trade-offs at a glance

<table>
<tr><td colspan="5">✓ = “pro” ✗ =”con” ☠= “infeasible or nearly infeasible” = strategy = proposal</td></tr>
<tr><td colspan="4">State Machine as a First Class Object
Coroutine frame is exposed as a type</td><td>Hidden State Machine
Coroutine frame type and layout hidden from program</td></tr>
<tr><td colspan="4">✓ Allows application to place coroutine frame as local variable on stack, as a class member</td><td>✓ Minimal type system impact</td></tr>
<tr><td>Early-Split
Compiler front-end calculates layout based on analysis of the source</td><td colspan="4">Late-Split
Size/layout of the frame is calculated in a later compilation phase after (some) optimisations have been performed</td></tr>
<tr><td rowspan="4">Unified Proposal (option B), Symmetric Coroutines(?), Resumable Expressions(?)
✓ constant sizeof available for coroutine state
✗ Inhibits optimisations of function across suspend-points in at least some (possibly many) cases.
☠Excessively expensive in some current frontends (would require major reengineering)</td><td colspan="4">✓ More optimization opportunities
✗ Difficult/impossible to get accurate sizeof at semantic analysis time</td></tr>
<tr><td>High Sizeof
A conservative estimate of coroutine frame size is made early</td><td>Late-sized types
Classes can end with a coroutine whose layout is determined late</td><td>Deferred layout types
Layout decisions are in general deferred until after coroutine split</td><td rowspan="2">✗ Frame size is not a constant at semantic analysis time, so dynamic allocation is required (but may be optimized out)</td></tr>
<tr><td>(no proposal)</td><td>Core Coroutines</td><td>Unified Proposal (option C)</td></tr>
<tr><td>✓ sizeof is a constant expression
✗ Could significantly overestimate than what is actually required
✗ Some optimizations need to be disabled
☠Unclear that it would be feasible to compute a reliable upper bound</td><td>✗ sizeof is not a constant at semantic analysis time
✓ But it is accurate, if provided
✗ Can only be embedded as a last field of a structure; makes the enclosing type late-sized too
✓ Offsets of fields are still constant
✗ ODR problems if a coroutine type is used across translation units and layout is exposed</td><td>✗ sizeof is not a constant at semantic analysis time
✓ But it is accurate, if provided
✓ Can be embedded as a field anywhere in the structure; makes the enclosing type deferred layout type too
✗ Field offsets not constant at semantic analysis time
☠Significant increase in implementation complexity even compared to late-sized types</td><td>Coroutines TS, Unified Proposal (any option)
✗ Fixed wrapping algorithm with many customization points
✓ Fits current compiler structure
✓ Can perform coroutine-to-coroutine tail calls without language changes</td></tr>
</table>

3 Coroutine Usability

The coroutine state machine object (like the activation record for a regular function) is by necessity immovable and uncopyable[3].

A coroutine state machine object is often not useful by itself and requires wrapping into a class with higher level semantics. Usually, a type-erased wrapper that involves heap allocation and type erasure **or** an embedded wrapper which wraps the immutable coroutine objects into a semantically meaningful immovable class and in case of symmetric coroutines the behaviour is customized via subclassing the base coroutine class.

The following is an illustration of user facing syntax for both (using Core Coroutines syntax).

Type-erased coroutine	Embedded coroutine (expected to be simpler in r3)

`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 r3 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 (or expose it as named class as in Symmetric Coroutines proposal).

Type-erased coroutines	Embedded coroutines
✓ Can be forward declared ✓ Can be called recursively ✓ Can be put on ABI boundary ✓ Does not require solution to sizeof challenge ✓ Definition can reside in the implementation file ✓ Can be virtual ✗ BUT!!! Requires heap allocation & indirect calls	✗ Cannot be forward declared ✗ Cannot be called recursively ✗ Cannot be put on ABI boundary[4] ✗ Requires solution to a sizeof challenge ✗ Definition must reside in the header / module interface file ✗ Cannot be virtual ✓ Does not require heap allocation

[3] Producing relocatable stack frames might be fun, but it's not part of any existing proposal.

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

Coroutines TS offers only type-erased coroutines at the moment. While non-type-erased coroutines can be added to any type-erased coroutine solution as an (essentially) separate mechanism, authors of this paper do not have consensus whether non-type-erased coroutines can be added to the Coroutines TS design while maintaining a single coherent interface to the coroutines mechanism.

We agree that finishing the design of Core Coroutines as an alternative to Coroutines TS is not expected to solve the same set of use cases as Coroutines TS with the same end-user convenience and the same performance (as in less convenient and less performant for some use cases, see use cases document).

4 Compiler-based wrapping vs Library-based wrapping

The Coroutines TS **today** does not expose the coroutine state machine object directly and uses compiler-based wrapping of the coroutine state machine guided by a set of customization points.

The Core Coroutines and Symmetric Coroutines proposals, on the other hand, provide a manifestation of the coroutine state machine as a function object that allows implementing most of the coroutine wrapping algorithm in the library.

Compiler based type-erasure	Library based type-erasure
✗ Wrapping algorithm fixed by core language ✗ Has more customization points ✓ Allocation can be optimized out ✓ Indirect calls can be replaced with direct calls ✓ Can be inlined (after devirtualization) ✓ Does not require solution to a sizeof problem	✓ Flexible wrapping algorithm ✓ Fewer customization points ✗ Substantially harder to optimize out heap allocation ✗ Substantially harder devirtualize indirect calls ✗ Substantially harder to inline (because needs devirtualization) ✗ Requires solution to a sizeof problem

The Unified Coroutines proposal provides an incremental improvement to the Coroutines TS design to allow a library to control the algorithm for constructing the coroutine frame rather than using a hard-coded algorithm that is controlled via customization points, while still retaining the compiler-based type-erasure. This allows the library to defer creation of the coroutine frame until later.

5 Early-split vs Late-Split

Early Split

Early-Split is a shorthand for the coroutine transformation performed at the front-end time before any optimizations applied to the coroutine body and therefore its representation as an object can be considered a complete type with sizeof being a constant expression at semantic analysis time.

Good Points: The early split produces an accurate size of a coroutine state machine, and a human reader of the code can reason about the size of the coroutine frame in a similar way to how they would reason about the size of a struct. However, the coroutine frame may potentially be bigger than in the late-split approach (where the optimizer would be able to simplify the body of the coroutine before the transforming it into a state machine), if the programmer does not optimize it "by hand" by carefully arranging what local state is live across suspension points.

Bad Points: Early split does not take advantage of an ever-growing set of transformations produced by the emergent complexity of increasingly-sophisticated compiler optimization techniques, which both improve coroutine efficiency and/or reduce its footprint. For example:

```
[[gnu::pure]] // annotated or inferred
int compute_value(std::array<char, kAsLargeAsYouLike> &);

generator<int> f() {
  co_yield 123;
  std::array<char, kAsLargeAsYouLike> arr;
  co_yield compute_value(arr);
}
```

Note that the lifetime of the array extends across the second co_yield, but the optimizer can tell that the array is never accessed after the call to compute_value because that function is pure, so the array does not need to be part of the suspension state of the coroutine.

Similar things happen with constant propagation, with inlining removing the last escape of a local variable, etc, etc.

Other optimizations that might remove heap allocations, eliminate suspends points reducing code size and increasing run-time efficiency are not available for early-split coroutine transformation. Early Split is essentially syntactic sugar avoiding the need to manually create a coroutine state machine, though in this approach compiler unlikely to match the efficiency of a hand-crafted state machine.

Does not fit traditional frontend structure for a frontend structured with distinct Parser, Semantic Analysis, Codegen Preparation steps.

Codegen preparation inserts handling of cleanups, destructors, exception handling and may introduce extra temporaries that will need to be included into the coroutine state. While it is theoretically not an insurmountable challenge, it might be a **major re-engineering** effort to the front-end structure and experts on two compiler front ends (Clang, EDG)[5] have indicated that this is not a practical approach.

[5] MSVC and GCC experts agree that significant restructuring might be required, but have not yet reached a conclusion how drastic it is.

The section describing High-Sizeof approach will cover the implementation complexity in more details, since early-split shares some of its implementation complexity with high-sizeof approach.

Late Split

Late-Split is a shorthand for the coroutine transformation performed in the middle-end or a back-end of the compiler. A key property of the late-split is that it allows to delay the transformation until the function body was simplified through various optimizations in the middle-end of the compiler.

This technique is easier to fit in the modern compiler structure and results in coroutines that are optimizable, however obtaining an accurate sizeof during semantic analysis is extremely challenging, and the amount of memory actually allocated for the coroutine state may be smaller or larger than the programmer had anticipated.

It is highly desirable that the implementation have as much control over the splitting actions as possible; for example the LTO pipelines are different in LLVM and GCC, and the optimal points for carrying out splitting are almost certainly going to be placed differently. Pre- and post-split optimizations may be applied, ignorant of the coroutine nature of the function.

Note, current implementations using type-erasure represent the size of the coroutine state as an internal compiler intrinsic, until the point at which it is known (by completing the coroutine body transform), this constant is then fed into all users of the intrinsic, and constant folded etc. to optimize those uses.

6 Schemes for dealing with unknown object sizes

We recognise that one of the primary and significant challenges in exposing the coroutine state object is related to the fact that its size is indeterminate in the face of optimisations and some transformations typically carried out in support of exception handling.

To that end, there are a number of early suggestions about how this might be handled in the compiler. These broadly move between “keep the size ‘known’ early (i.e. estimate it)” to “modify the behaviour of the compiler so that a complete type is permitted to have unknown size” at, or up to, various points in the compilation pipeline.

7 High-Sizeof

High-Sizeof approach refers to the technique where we allow the coroutine transformation to happen late after the coroutine body is simplified, but, we are making an estimate of the size required for the additional content needed for the coroutine state machine at semantic analysis time. In this way, the size of the object is fixed and known (we stay within the expectations of the compiler for complete types).

A likely limitation is that such approach may highly overestimate the required size. It is possible to construct examples for the coroutine state to be 10,000x higher than it should be, though we expect that in ordinary programming the overestimation will be on the order of 3-10x.

Another hazard of the High-Sizeof estimate that some of the optimizations might increase the size require for the coroutine frame, such as promoting small allocations to stack temporaries and heap allocation elision optimizations. To work with high-sizeof approach those optimizations need to be disabled for coroutines which are using high-sizeof estimate. Auditing optimizations to determine whether they can be safely run will be significant effort. There are cases where the only safe action is to disable them. It is similar to the tension between holding more state locally, and running out of registers. Except that in this case we arrive at a failure to compile, rather than spilling to a stack frame (and potentially degrading performance).

Frontend also has to make provisions for data inserted by sanitizers and other tooling.

It is likely that there will be a compile time check in the middle end to confirm that the estimate is sufficiently large for the coroutine state to fit in and report compile time error otherwise. To workaround, it would be necessary to have a pragma that allows to increase per-file or per-function estimate to make the program compile. Users will inevitably use this pragma to select exactly the right size, resulting in very brittle programs that fail to compile at the slightest source change, differing options, or compiler update.

The high-sizeof approach requires the semantic analysis stage of the compiler to have knowledge of how the state of the coroutine will initially be represented (prior to any optimizations being applied). As a simple example, consider a program such as:

```
generator<int> f(X v) {
  Y arr[10];
  const X &x = some_condition ? v : X(1, 2, 3);
  co_yield 42;
  co_yield g(arr, x);
}
```

Here, the coroutine state must contain sufficient space not only for the parameter v and the local reference x, but also potentially for invisible state indicating whether x.~X() must be run at the end of the coroutine body, which depends on whether some_condition was true on the second line. Additionally, the mechanism for emitting destructors for arr may need invisible state indicating how many elements of arr have been constructed (this is primarily of interest if an exception is thrown part-way through constructing the array, but for some implementations the same state might be reused for normal destruction when the variable goes out of scope). Whether such state is necessary and how large it is will vary based on conditions that semantic analysis traditionally has not needed to know or care about, so high-sizeof will need to make a conservative guess.

Essentially, we're saying here that we identify the user's exposed variables and then we allocate some bucket of memory that the optimiser can have to do what it wants with (and to hard fail if it can't work within that). From an implementation point of view, that seems achievable but likely to be quite an invasive change to optimisations (although some most likely already have parameterisation of allowable growth etc. equally likely, different between different vendors). As with any case where the user's variables are exposed in the frame, it ties the compiler's hands with respect to elision of such.

It is worth noting that the complexity necessary to implement high-sizeof without ever guessing too low is essentially the same work that is required to implement an early split. The key difference is that with high-sizeof, layout optimizations are still permitted on the output of early split so long as they do not make the coroutine state larger than it was initially.

8 Late-sized types

Late sizing treats the coroutine state as a trailing array of a structure, where the size is determined at some later point of the compilation process — unlike the current meaning of the “flexible array member”, where the size is potentially undetermined until run time.

As soon as a late-sized type is embedded as a last member of a structure, the type of the enclosing structure becomes a late-sized type as well. Thus, late-sized types have a non-trivial impact on the type system. Late-sized types cannot be base classes. The sizeof operator applied to a late-sized type cannot be a constant expression: It must instead be either a run-time value or ill-formed.

If the size is exposed at run time, a further issue emerges: either the size of the type can vary between translation units, leading to further type system problems and/or ODR issues, or the size (and layout) of the type is the same across all translation units, hindering the ability to optimize (and, in particular, partially inline) the coroutine.

9 Deferred-layout types

If deferred layout types are to become full members of the type system, the backend of the compiler becomes capable of driving template instantiation. Current implementations of link time optimizations use a different compiler than the C++ parser — more akin to a compiler reading a “link-time” language — and therefore the template machinery is not available. Furthermore, even if the template implementation were available, the source representation needed for instantiation is not retained past the C++ front end’s hand-off of the IR to the middle end. The IR is incapable of representing the necessary information. For instance, type information may well have been obscured, by placing base classes as regular members of their containing classes.

Reorganizing a compiler to allow this kind of feedback loop would be a person-decade level of effort or more. It would be invasive throughout the compiler, resulting in pushback from other members of the compiler development community, all having to pay a cost for a language they might have no interest in.

Further there is simply the specification problem of which entities such late-invoked instantiations might see -- just those from the originating source (which, the coroutine’s definition, the direct caller it is inlined into, the ultimate container of an inline nest?). These questions have proved vexing for the modules development[6].

At some point of lateness, we reach the status quo - where the [variable part of] the frame layout is fully under control of the optimisation process (i.e. we have the same freedom now applied to objects as was previously used only for the activation record). However, we are still unable to elide exposed user variables, even if some optimisation or transform makes them unused.

10 Guaranteed Tail-call convention

Coroutine transformation may convert loops into unbounded recursion leading to stack exhausting. Core Coroutine proposed a partial solution that does not yet work for asynchronous coroutines and coroutines defined across translation units and ABI boundaries.

[6] It’s perhaps amusing to note that this very thinking is what got us “export” templates: A vendor (SGI) had claimed that they had a well-advanced plan for template instantiations performed as a step of the back end and even as part of the dynamic linker/loader. Those plans never resulted in demonstrable products.

While no detail design work was done yet, it is likely that a solution would involve adding a guaranteed tail-call calling convention to the C++ language and the type system. Functions that contain “tail return” statement are likely to require declaring them as tail-callable as well. Such changes are required so that tail-callability becomes part of the function type and the property of tail-callability is preserved across indirect calls and across TUs..

Additionally, it is likely that all (most) platform-specific compiler backends would have to be updated to correctly handle the new calling convention.

At least for GCC, tailcalling is discovered during optimization, and marked explicitly (by the back end) at the calling point. Providing the user with the ability to specify a tail call, will require some error checking, along the lines described in P1063, to ensure a tail call is well formed (generally, the caller and callee require the same space for arguments and there are no local entities whose lifetime cannot be shortened to before the tailcall).

11 Symmetric Coroutine Definition Syntax

The Symmetric Coroutines proposal contains a novel syntax for declaring classes:

```
template <typename R, typename T> R f(T u) { do1(); u(); do2(); }
```

The definition above could be defining a function or a class, depending on what is the type R is. Clearly allowing this would raise new ambiguity concerns.

Note also that the invocation of u() may or may not represent a suspension point depending on what type T is.

12 Symmetric Coroutine Calling Convention

The Symmetric Coroutines proposal depends on a special calling convention. As specified, only operator() of types coroutine, derived from type coroutine and resume_continuation have this calling convention.

Invoking operator() from a coroutine suspends the current coroutine and tail-resumes the coroutine associated with the object operator() belongs to.

Implementing this calling convention would likely require changes to platform-specific compiler backends to correctly handle the new calling convention.

It is unclear from the Symmetric Coroutine paper whether this property “suspend-current-coroutine-and-tail-resume another one” is a property of the function type of the operator(), which would mean that taking an address or using a function object wrapper would retain it, or decayed to a normal function call.

If this new calling convention has to propagate further than the C++ front end, it will be invasive to implement. It is very concerning that the possibility of linker (and debugger/debug info) changes may be needed. Alternatively, if the semantics can be reduced to a set of regular {tail}call operations during the conversion of the C++ representation to the middle end IR, this is unlikely to be a burden.

13 Implementation

Coroutines TS has been implemented in publicly released compilers in MSVC (5 years), Clang (2 years and was available earlier in a public fork), EDG (4 years) and an in-progress (partial) implementation is available as a GCC public branch today.

Implementability of Core Coroutines and Symmetric Coroutines depends of feasibility of the proposed solutions to exposing the coroutine state object to a compiler at semantic analysis time, and the work required to implement new calling conventions guaranteed tail call convention or symmetric coroutine call convention as they require.

14 Conclusion

We consider that given current understanding of the compiler technology, there are in practice two feasible points in the coroutine design space.

1) late-split coroutine with hidden state machine object (the TS approach)
2) late-split coroutine represented as a type with a non-exposed layout (the Core coroutines approach)

Deferred layout types are a research problem that have uncertain language and implementation impact. Late-sized types with fixed upper layout bound are not obviously feasible (not are they formally proposed). Early split is not practical for at least two of the major front end implementations.

To get a first-class state machine from the first approach (e.g., the TS), requires manual sizing as described in part 3 of [P1362R0] or high-sizeof estimator. Both of these are possible today using existing compiler technology. Though the latter may have undesirable user experience as highlighted in the high-sizeof section.

Introducing late-sized types in the second approach (e.g., Core coroutines) has non-trivial impact on the language type system and does not achieve the same near-term[7] efficiency as the Coroutine TS approach for the type-erased asynchronous coroutines [P1362R0] and may not match expectations of the users hoping to get a “normal” object representing a coroutine.

References:

N4775	Working Draft, C++ Extensions for Coroutines	Gor Nishanov
P0981R0	Halo: coroutine Heap Allocation eLision Optimization: the joint response	Richard Smith, Gor Nishanov
P1063R2	Core Coroutines	Geoff Romer, James Dennett, Chandler Carruth

[7] In the longer term (5+ years), equivalent performance gains may be achievable via non-coroutine-specific optimizer technology and language features.

P1342R0	Unifying Coroutines TS and Core Coroutines	Lewis Baker
P1362R0	Incremental Approach: Coroutine TS + Core Coroutines	Gor Nishanov
P1477R0	Coroutines TS Simplifications	Lewis Baker
P1430R0	First-class symmetric coroutines in C++	Mihail Mihaylov, Vassil Vassilev

Appendix

Algorithm for creating coroutine frame

Hard-coded Algorithm Coroutines TS	Optional Hard-coded Algorithm Core Coroutines	Customisable Algorithm Unified Coroutines
Compiler allocates coroutine frame unconditionally when coroutine is called. Hard-coded algorithm calls methods on the promise type. - promise_type::operator new - copying/capturing parameters - promise constructor - get_return_object() - initial_suspend() - coroutine_hande::destroy()	Provides access to coroutine lambda primitive that lets you write algorithm to wrap/create coroutine yourself. Using the “sugar syntax” maps to a different hard-coded algorithm. Construct return-type, passing a lambda which when invoked returns the coroutine lambda/frame object. More parts of the algorithm are under library control. - Heap allocation or not - Type erasure or not Allows deferring creation of coroutine frame until after the call. Library responsible for type erasure of coroutine frame. Some open problems to solve with sugar syntax to avoid needing to type-erase coroutine frame.	Invocation of coroutine function is directly translated to a call to get_return_object() that is passed a factory function that constructs coroutine state. Allows deferring creation of coroutine frame. Allows deferring choice of promise_type used to instantiate coroutine. Allows customising allocation - compiler-type-erased + heap allocated or first-class type. Allows customising return-type - does not need to be the same as the wrapper-type. Default algorithm based on function signature if you don’t specify. Allows optionally overriding the algorithm independently of the

		function signature.

A very nice table, most of the content was absorbed in the Trade-offs at a glance table, but it was a shame to delete this one.

<table>
<tr><th>Hidden Frame Type
Coroutine frame type and layout hidden from program.</th><th colspan="3">Expose Coroutine Frame as a Type</th></tr>
<tr><td>Coroutines TS

Fixed wrapping algorithm with customization points</td><td colspan="3">Coroutine frame is exposed as a type.

Allows application to place coroutine frame as local variable on stack, as a class member.

Requires a solution to the “sizeof” problem: various options below.</td></tr>
<tr><td rowspan="3">An opaque handle to the frame is exposed to the program.

Frame is heap-allocated by the compiler. Allocation may be customisable by application.

Compiler is allowed to elide the allocation by inlining into the frame of the caller.</td><th>Early-Sized Type</th><th colspan="2">Late-Sized Type</th></tr>
<tr><td>Compiler front-end calculates layout based on analysis of the source.

sizeof(T), alignof(T) and offsetof(T, member) are all constexpr

All variables/temporaries that span a suspend-point are placed in coroutine frame.

Inhibits optimisations of function across suspend-points in at least some (possibly many) cases.

Allows stable ABI/layout for inline coroutine functions.

Can allow coroutine frame objects to be passed between TUs.

Computing a conservatively-correct upper bound for the frame size would be excessively expensive in some current frontends (would require major reengineering) and hard to specify in ABI.</td><td colspan="2">Size/layout of the frame type is calculated in a later compilation phase.

sizeof(T), alignof(T) are constant but not constexpr, or sizeof(T), alignof(T) are ill-formed.

Allows compiler to calculate the layout of the coroutine frame after (some) optimisations have been performed.

Frame layout for inline coroutines may be calculated differently in each TU it is used in, resulting in ODR problems if the layout is exposed to the program.

Cannot forward-declare/pass coroutine frame objects between TUs without type-erasure.</td></tr>
<tr><td></td><th>Hidden-Layout Type</th><th>Deferred-Layout-Type</th></tr>
<tr><td></td><td>Hidden-layout type is only allowed in tail-position (not in a base class, not in an array, not in a class with virtual base classes)

Offsets of other data members of a class with a coroutine member are still constant expressions.</td><td>Allows composing structures containing multiple deferred-layout objects.

offsetof(T, member) is not constexpr

Severe implementation burden in some implementations.</td></tr>
</table>