Benefits of Frame-Visible Coroutines for Senders

Document Number:	P4166R0
Date:	2026-05-01
Intent:	Inform
Audience:	LEWG

Reply-to:	Vinnie Falco vinnie.falco@gmail.com C++ Alliance Proposal Team

Table of Contents

Abstract

Frame-visible coroutines would eliminate `std::execution::task`'s heap allocation and give sender-based coroutine algorithms full optimizer visibility.

The sender three-channel completion model is a genuine achievement: compile-time verification of async completion paths and generic algorithms that dispatch on completion disposition without knowing the concrete sender type. A structural constraint exists - compound I/O results cannot round-trip through three mutually exclusive channels - and two of the three properties that senders currently provide beyond the coroutine model trace to one language choice: the coroutine frame is opaque. P1492R0 [1] documented a Known-Layout Type model where the frame becomes a first-class type with `constexpr` size and alignment. If C++ added frame-visible coroutines alongside the existing opaque-frame model, `std::execution::task` eliminates its heap allocation, coroutine-based sender algorithms gain full optimizer visibility, and both async models that

C++26 ships improve. Compile-time work graphs - the type-level encoding of dependency topology - remain the unique sender contribution.

Revision History

R0: May 2026 (pre-Brno mailing)

- Initial version.

---

1. Disclosure

The author provides information and serves at the pleasure of the committee.

This paper is part of the Network Endeavor (P4100R0 [2]), a project to bring coroutine-native I/O to C++.

The author developed and maintains Corosio [3] and Capy [4] and believes coroutine-native I/O is a practical foundation for networking in C++.

Coroutine-native I/O and `std::execution` are complementary. Each serves the domain where its design choices pay off.

The author regards `std::execution` as an important contribution to C++ and supports its standardization for the domains it serves well - GPU dispatch, heterogeneous execution, and compile-time work-graph composition among them. The three-channel completion model enables generic algorithms that dispatch on completion disposition without knowing the concrete sender type. Compile-time work graphs - the type-level encoding of dependency topology in the sender operation state - are a sender property the coroutine model does not replicate.

The author believes that C++ would benefit from having all three coroutine models in the language simultaneously: fibers (stackful), frame-opaque coroutines (C++20), and frame-visible coroutines (previously proposed in P1492R0 [1] and P1342R0 [5]). Each covers non-overlapping use cases, and users are better served by having access to all of them. Frame-visible coroutines would benefit both async models that C++26 ships.

This paper asks for nothing.

---

2. The Three-Channel Achievement

The sender model provides a universal completion protocol. Every sender completes through exactly one of three channels (P2300R10 [6] Section 5.1):

- `set_value(Ts...)` signals successful completion with zero or more result values.
- `set_error(E)` signals that an error occurred during scheduling, execution, or at an earlier point in the sender chain.

- `set_stopped()` signals that the operation completed without succeeding and without failing - typically because cancellation was requested.

Completion signatures declare these channels as type-level contracts. A sender that declares `completion_signatures<set_value_t(int), set_error_t(error_code), set_stopped_t()>` has told the compiler every way it can complete. A type mismatch between pipeline stages is a compile error. This is a genuine achievement: compile-time verification of async completion paths.

---

3. Compound Results in the Three-Channel Model

Route A passes the compound result through the value channel. P2300R10 writes:

> *"Often in the case of partial success, the error condition is not fatal nor does it mean the API has failed to satisfy its post-conditions. It is merely an extra piece of information about the nature of the completion. In those cases, 'partial success' is another way of saying 'success'. As a result, it is sensible to pass both the error code and the result (if any) through the value channel."*

The specification provides a code example:

```
return read_socket_async(socket, span{buff})
    | execution::let_value(
        [](error_code err, size_t bytes_read) {
            if (err != 0) {
                // partial success
            } else {
                // full success
            }
        });
```

The `error_code` travels on `set_value`. `upon_error` does not see it. Generic error-handling algorithms are unreachable for this error.

Route B passes the compound result through the error channel. P2300R10 writes:

> *"In other cases, the partial success is more of a partial failure. ... In those cases, sending the error through the value channel loses valuable contextual information. It's possible that bundling the error and the incomplete results into an object and passing it through the error channel makes more sense. In that way, generic algorithms will not miss the fact that a post-condition has not been met and react inappropriately."*

The byte count is now inside an object on the error channel. Value-channel algorithms cannot see it.

P2300R10 also proposes a third workaround - returning a *range* of senders:

> *"Another possibility is for an async API to return a range of senders: if the API completes with full success, full error, or cancellation, the returned range contains just one sender with the result. Otherwise, if the API partially fails ... the returned range would have two senders."*

Two senders for one I/O operation.

The normative text (P2300R10 [6] Section 34.2) defines the constraint:

> *"once started, eventually completes exactly once with a (possibly empty) set of result datums and in exactly one of three dispositions: success, failure, or cancellation."*

A socket read that transfers 500 bytes and then encounters `ECONNRESET` has both an error condition and a byte count. Both are produced. Both are meaningful. One disposition does not describe the completion.

The C++26 working draft ships `stopped_as_error` (P2300R10 [6] Section 4.20.8), an adaptor that collapses the stopped channel into the error channel. The standard provides a mechanism to reduce three channels to two.

---

4. The Forward Mapping

An awaitable return type is a type that supports structured bindings through the tuple protocol (`std::tuple_size`, `std::tuple_element`, and `get`). io_result [4] is an example:

```
template<class... Ts>
struct io_result
{
    std::error_code ec;
    std::tuple<Ts...> values;
};
```

With `std::tuple_element<0, io_result<Ts...>>` specialized to `std::error_code`.

A compile-time trait maps any such return type to the sender three-channel routing:

```
template<typename Result>
struct has_error_channel : std::false_type {};

template<typename Result>
    requires (std::tuple_size_v<Result> >= 1)
        && std::is_same_v<
            std::tuple_element_t<0, Result>,
            std::error_code>
struct has_error_channel<Result> : std::true_type {};
```

The trait inspects the return type's first element at compile time. If it is `std::error_code`, the return type carries an error channel. The remaining elements are the value payload. Cancellation propagates through `io_env`'s stop token, out of band.

The mapping covers every case:

Awaitable return type	Error channel	Value channel	Stopped channel
`io_result<size_t>`	`ec`	`size_t`	stop token
`io_result<size_t, metadata>`	`ec`	`size_t, metadata`	stop token
`io_result<>`	`ec`	(none)	stop token
`T` (no `error_code` at index 0)	(none)	`T`	stop token

Every row maps to a sender `completion_signatures`. The trait compiles. The mapping is total.

---

5. The Reverse Mapping

Given sender completion signatures, reconstruct the awaitable return type:

```
// Given:
//   completion_signatures<
//       set_value_t(size_t),
//       set_error_t(error_code),
//       set_stopped_t()>
//
// Reconstruct: io_result<size_t>
//   where ec and n are BOTH always produced
```

The reconstruction requires that when `set_error(ec)` fires, the `size_t` from `set_value` is also available. It is not. The three-channel model guarantees exactly one channel fires per completion. When the error channel fires, the value channel's data does not exist.

An I/O read that transfers 500 bytes and then encounters a connection reset produces both an `error_code` and a byte count. In the awaitable model:

```
auto [ec, n] = co_await stream.read_some(buf);
// ec == connection_reset, n == 500
// both values present, both inspectable
```

In the sender model, the operation must choose:

```
// Option A: set_value(ec, n)
//   upon_error cannot see ec
//   retry does not trigger
//   the error is invisible to generic algorithms

// Option B: set_error(ec)
//   n is lost
//   the 500 bytes are gone
//   the caller cannot use partial data
```

Neither option preserves the compound result. The mutual exclusion of channels is the structural cause. The forward mapping (Section 4) is injective: distinct awaitable return types map to distinct channel configurations. The reverse mapping is not a function: multiple awaitable return types (one with compound results, one without) map to the same channel configuration, and the channel configuration cannot distinguish them.

The round-trip `awaitable -> channels -> awaitable` loses data. The mapping works one direction.

---

6. The Frame Opacity Cost

The sender model currently provides three properties that coroutines do not:

- **Full pipeline visibility to the optimizer.** The sender operation state is parameterized on the receiver type. The compiler sees the full pipeline before `start()` is called. It can inline across operation boundaries, eliminate dead code, and propagate constants. The coroutine frame is opaque behind `coroutine_handle<>`. The optimizer cannot see through `resume()`.
- **Zero-allocation composition.** The sender operation state can be stack-allocated or embedded in the parent's operation state because its type is fully known at compile time. The coroutine frame must be heap-allocated because its size is determined after inlining and optimization passes. No compiler guarantees HALO. P3552R3 [7] `std::execution::task` heap-allocates for this reason - the sender operation state cannot embed a coroutine frame whose size is unknown at compile time.
- **Compile-time work graphs.** The sender pipeline type encodes the dependency topology at the type level. `when_all(s1, s2) | then(f)` produces a type from which a scheduler can determine at compile time that `s1` and `s2` run concurrently and `f` depends on both. A coroutine's internal control flow is imperative. No type-level inspection can recover the dependency structure from sequential `co_await` statements.

The first two properties trace to one cause: the coroutine frame is opaque. `coroutine_handle<>` erases the frame's type. The sender model stamps the receiver into the operation state, making the type visible. This is a design choice made in C++20, not a permanent constraint.

The third property traces to the composition model itself. Sender composition is declarative: the programmer describes a computation graph, and the type system encodes it. Coroutine composition is imperative: the programmer writes sequential statements, and the control flow executes them. Making the frame visible does not make the control flow declarative.

---

7. Frame-Visible Coroutines

P1492R0 [1] "Coroutines: Language and Implementation Impact" (Smith, Vandevoorde, Romer, Nishanov, Sidwell, Sandoe, Baker, 2019) documented three competing coroutine frame models during the C++20 standardization. The committee chose the opaque-frame model. The Known-Layout Type model was documented but not adopted. It provides:

> *"`sizeof(T)`, `alignof(T)` and `offsetof(T, member)` are all `constexpr`. ... Allows stable ABI/layout for inline coroutine functions. Can allow coroutine frame objects to be passed between TUs."*

P1342R0 [5] "Unifying Coroutines TS and Core Coroutines" (Baker, 2018) described the concrete mechanism: a coroutine lambda whose frame is a first-class type in the type system, allowing the caller to `sizeof` it, stack-allocate it, or embed it in a

struct.

If C++ added a frame-visible coroutine type alongside the existing opaque-frame coroutines, `std::execution::task` would no longer need to heap-allocate. The sender operation state could embed the coroutine frame directly, because `sizeof(frame)` is now `constexpr`. The compiler sees through the coroutine boundary - the same optimizer visibility that sender pipelines provide today. Both models improve:

Property	Opaque-frame	Frame-visible	Senders
Full pipeline visibility to optimizer		Yes	Yes
Zero-allocation composition		Yes	Yes
Compile-time work graphs			Yes
Language control flow (`for`, `if`, ...)	Yes	Yes	
User-defined N-channel return types	Yes	Yes	
Compound result preservation	Yes	Yes	
Type-erased streams at zero per-op cost	Yes		
Separate compilation	Yes		
ABI stability	Yes		

The sender column gains two properties it currently lacks when coroutine-based sender algorithms are written with frame-visible coroutines: language control flow and compound result preservation. The frame-visible column gains two properties that only senders provide today: optimizer visibility and zero-allocation composition. Compile-time work graphs - the type-level encoding of dependency topology - remain unique to senders. Making the frame visible does not make imperative control flow declarative.

---

8. What Each Model Gains

Senders gain coroutine control flow without losing performance. A sender algorithm implemented as a frame-visible coroutine has `for`, `if`, `break`, `return`, structured bindings, and `co_await` - language constructs that today require choosing between coroutine ergonomics and sender performance properties. Frame-visible coroutines remove that choice. `std::execution::task` eliminates its heap allocation because the sender operation state can embed the frame. Coroutine-based sender algorithms gain optimizer visibility because the compiler sees through the frame boundary.

Coroutines gain optimizer visibility without surrendering their properties. Frame-visible coroutines preserve language control flow, user-defined return types (Section 4), and compound result preservation (Section 5). They add the two properties that only senders provide today: full pipeline visibility to the optimizer and zero-allocation composition.

Opaque-frame coroutines remain available for the properties they uniquely provide - type-erased streams, separate compilation, and ABI stability.

Compile-time work graphs remain the unique sender contribution. The programmer describes a computation graph and the type system encodes it. `when_all(s1, s2) | then(f) | transfer(gpu_scheduler)` produces a type from which a scheduler can determine - at compile time, before any work begins - that `s1` and `s2` run concurrently, that `f` depends on both, and that the result transfers to a GPU context. The scheduler sees the full dependency topology because the pipeline type is the dependency topology. Making the frame visible does not make imperative control flow declarative. This property is irreducible to the sender composition model and defines the domain where declarative lazy composition is the right tool.

---

9. Conclusion

Frame-visible coroutines would eliminate `std::execution::task`'s heap allocation and give both async models optimizer visibility. The forward mapping (Section 4) and reverse mapping (Section 5) identify a structural constraint in the three-channel model - compound results cannot round-trip through mutually exclusive channels - that frame-visible coroutines help bridge by allowing coroutine-based sender algorithms to preserve compound results while retaining the performance properties of sender composition. Compile-time work graphs remain unique to senders and define the sender model's irreducible contribution. P1492R0 [1] deserves renewed attention because the language addition it documented would improve both async models that C++26 ships.

Acknowledgments

The author thanks Eric Niebler, Kirk Shoop, Lewis Baker, and their collaborators for `std::execution` and the three-channel completion model; Andrzej Krzemieński for identifying compound success as a general programming problem; Ville Voutilainen for confirming the dispatch gap on the LEWG reflector; Peter Dimov for the refined channel mapping; Gor Nishanov for the coroutine model's explicit support for task type diversity; and the authors of P1492R0 [1] for documenting the design space that includes frame-visible coroutines.

References

[1] P1492R0 - "Coroutines: Language and Implementation Impact" (Richard Smith, Daveed Vandevoorde, Geoffrey Romer, Gor Nishanov, Nathan Sidwell, Iain Sandoe, Lewis Baker, 2019).

[2] P4100R0 - "The Network Endeavor: Coroutine-Native I/O for C++29" (Vinnie Falco, Steve Gerbino, Michael Vandeberg, Mungo Gill, Mohammad Nejati, 2026).

[3] cppalliance/corosio - Coroutine-native networking library.

[4] cppalliance/capy - Coroutine I/O primitives library.

[5] P1342R0 - "Unifying Coroutines TS and Core Coroutines" (Lewis Baker, 2018).

[6] P2300R10 - "`std::execution`" (Eric Niebler, Michał Dominiak, Lewis Baker, Kirk Shoop, Lucian Radu Teodorescu, Lee Howes, 2024).

[7] P3552R3 - "Add a Coroutine Task Type" (Dietmar Kühl, Maikel Nadolski, 2025).