Combinators and Compound Results from I/O

Document Number:	P4124R0
Date:	2026-05-01
Intent:	Inform
Audience:	LEWG, SG1

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

Table of Contents

Abstract

Revision History

- R0: May 2026 (pre-Brno mailing)

1. Disclosure
2. What when_all Should Do for I/O
   - 2.1 io_result
   - 2.2 Behavior
   - 2.3 Return Type
3. The Problem
   - 3.1 The Specification
   - 3.2 Three Strategies
   - Strategy A: Value Channel Only
   - Strategy B: Error Channel with Tuple
   - Strategy C: Predicate on when_all
   - 3.3 Convergence
4. The Adapter
   - 4.1 Coroutine Adapter
   - 4.2 Struct Adapter
5. The Cost of the Adapter
   - 5.1 Allocation
   - 5.2 Call-Site Boilerplate
   - 5.3 Code Size
   - 5.4 The "Write It Once" Inversion
6. Domain-Aware Combinators
   - 6.1 IoAwaitable Children
   - 6.2 Sender Children
   - 6.3 Dispatch
   - 6.4 Generalization

7. Comparison
8. Conclusion
Acknowledgments
References

Abstract

`std::execution::when_all` dispatches on channels. I/O errors arrive on the value channel. The combinator does not see them.

This paper traces the `when_all` completion handler through the P2300R10 [1] specification, examines three strategies for routing I/O compound results through the three-channel model, shows that all three fail to achieve correct error-driven cancellation, and proposes domain-aware combinators - a compile-time dispatch that selects the I/O-aware path when the children are I/O awaitables and delegates to `std::execution::when_all` when the children are senders.

Revision History

R0: May 2026 (pre-Brno mailing)

- Initial version

---

1. Disclosure

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

The author developed and maintains Corosio [2] and Capy [3] 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. Nothing in this paper argues for removing, delaying, or diminishing `std::execution`. A coroutine-only design cannot express compile-time work graphs, does not support heterogeneous dispatch, and assumes a cooperative runtime. Those are real costs.

This paper asks for nothing.

---

2. What `when_all` Should Do for I/O

The synchronous analogy governs the model:

```
std::tuple result{ f1(), f2(), ..., fN() };
```

If any of `f1`...`fN` fails, the caller gets the error. The tuple only exists on the success path.

An I/O-aware `when_all` follows the same principle. The combinator knows the result convention: the return type destructures with `error_code` in the first position. Success means `!ec`. Failure means `ec`.

2.1 `io_result`

`io_result<Ts...>` is a product type whose first element is `error_code` and whose remaining elements are `Ts...`. It supports structured bindings:

```
auto [ec, n] = co_await sock.async_read_some(buf);
```

`ec` is always present. `!ec` means success; the remaining elements are meaningful. `ec` means failure; the remaining elements are present but not guaranteed meaningful. `io_result` is defined in P4166R0 [9] and implemented in Capy [3].

2.2 Behavior

#	Scenario	Behavior
1	All children return `!ec`	Return tuple of all results. No cancellation.
2	One child returns `ec`, others pending	Cancel siblings. Propagate error.
3	Multiple children return `ec` concurrently	Each triggers stop (idempotent). First `ec` wins.
4	`ec == eof`, `n == 0`	Error. Cancel siblings.
5	`ec != 0`, `n > 0` (partial transfer)	Error. Cancel siblings. Values stored as-is. Caller sees `ec` and knows it failed; partial values are available but not guaranteed meaningful.
6	Zero-length buffer, `({}, 0)`	Success. No cancellation.
7	Zero-length buffer, `(ec, 0)`	Error (`ec` reflects stream state). Cancel siblings.
8	One child throws	Capture exception. Cancel siblings. Rethrow after all complete.
9	Multiple children throw	First exception captured. Others discarded. Rethrow first.
10	One throws, another returns `ec`	Exception wins. Tuple is not accessible.
11	Parent stop token fires	Stop requested for children. Children observing stop return `operation_aborted`; a child already completing may return its own result. First `ec` wins.
12	All children fail	Propagate single `error_code` (first wins). Not a tuple of failures.

The combinator inspects the **value** - the `error_code` in the first position of the result. It does not dispatch on a channel tag.

2.3 Return Type

Peter Dimov's design [5]: the return type lifts the `error_code` out of each child's result into a single outer `io_result`. Child value types are collected as parameters.

Given children returning `io_result<T1>`, `io_result<T2>`, ..., `io_result<Tn>`:

```
io::when_all(child1, child2, ..., childN)
  -> io_result<T1, T2, ..., Tn>
```

Three reads, each returning `io_result<size_t>`:

```
auto [ec, n1, n2, n3] = co_await io::when_all(
    stream.read_some(buf1),
    stream.read_some(buf2),
    stream.read_some(buf3));
```

One `ec`. Flat destructuring. On success, all three byte counts are meaningful. On failure, the first error wins and siblings were cancelled.

---

3. The Problem

3.1 The Specification

P2300R10 [1] specifies `when_all`'s completion logic in `impls-for<when_all_t>::complete`. The handler dispatches on the completion channel tag:

```
[]<class Index, class State, class Rcvr,
    class Set, class... Args>(
    this auto& complete,
    Index, State& state, Rcvr& rcvr,
    Set, Args&&... args) noexcept
    -> void
{
    if constexpr (
        same_as<Set, set_error_t>)
    {
        if (disposition::error !=
            state.disp.exchange(
                disposition::error))
        {
            state.stop_src.request_stop();
            TRY-EMPLACE-ERROR(state.errors,
                forward<Args>(args)...);
        }
    }
    else if constexpr (
        same_as<Set, set_stopped_t>)
    {
        auto expected =
            disposition::started;
        if (state.disp
                .compare_exchange_strong(
                    expected,
                    disposition::stopped))
        {
            state.stop_src.request_stop();
        }
    }
    else if constexpr (
        !same_as<decltype(State::values),
                 tuple<>>)
    {
        if (state.disp ==
            disposition::started)
        {
            auto& opt =
                get<Index::value>(
                    state.values);
            TRY-EMPLACE-VALUE(complete,
                opt,
```

```
                forward<Args>(args)...);
        }
    }

    state.arrive(rcvr);
}
```

Three branches. Three behaviors:

- `set_error`: calls `stop_src.request_stop()`. Siblings cancelled. Error stored.
- `set_stopped`: calls `stop_src.request_stop()`. Siblings cancelled.
- `set_value` (the else branch): stores values. No inspection of arguments. No stop request. No cancellation.

The `set_value` branch has no mechanism to inspect value-channel arguments, apply a predicate, or decide to cancel based on the values received. This is not an implementation choice. It is the normative specification. The dispatch is on the **channel tag**, not on the **value**.

3.2 Three Strategies

An I/O read returns `(error_code, size_t)`. Three strategies for routing this through the sender protocol. All three fail to achieve the behavior specified in Section 2.

Strategy A: Value Channel Only

Route `(ec, n)` through `set_value`:

```
co_await std::execution::when_all(
    sock.async_read_some(buf1),   // set_value(ec, n)
    sock.async_read_some(buf2));  // set_value(ec, n)
```

`when_all` stores both results. If `buf1`'s read fails with `connection_reset`, `when_all` does not know. `buf2`'s read runs to completion. The error is inside the value. The combinator is blind to it.

The entire error-handling algebra of senders - `upon_error`, `retry`, `let_error` - is useless for I/O under this strategy because errors never reach the error channel. The three-channel model is reduced to one channel.

Strategy B: Error Channel with Tuple

Route failures through `set_error(tuple(ec, n))`, successes through `set_value(n)`:

```
co_await std::execution::when_all(
    io_adapt(sock.async_read_some(buf1)),
    io_adapt(sock.async_read_some(buf2)));
```

`when_all` sees `set_error` on failure and cancels siblings. Correct for `when_all` specifically. Five problems remain:

1. **Someone must do the split.** The I/O operation returns one result. Something upstream must inspect `ec` and route to the correct channel. That is the `io_adapt` adapter examined in Section 4.
2. **The error type changes.** `set_error` now carries `tuple<error_code, size_t>`, not `error_code`. Every downstream `upon_error`, `let_error`, and `retry` must handle both `tuple<error_code, size_t>` and `exception_ptr`. The `if constexpr` guards are not optional.
3. **Byte count is semantically misplaced.** Partial transfer data - bytes successfully read before the error - is inside `set_error`. Downstream handlers that want to use those bytes must extract them from an error tuple. Data became part of an error.
4. `retry` **loses context.** `retry` sees `set_error(tuple(ec, n))` and retries the whole operation. The byte count inside the error tuple is lost. The retry does not know to resume from offset `n`.
5. **Composition breaks at boundaries.** A generic `upon_error` written for `error_code` does not compile against `tuple<error_code, size_t>`.

Strategy C: Predicate on `when_all`

Add a predicate parameter that inspects `set_value` arguments:

```
co_await when_all(
    [](auto const& result) {
        return !std::get<0>(result);
    },
    sock.async_read_some(buf1),
    sock.async_read_some(buf2));
```

Three problems:

1. **Changes the interface.** A predicate overload is a new algorithm that must be specified, reviewed, and standardized. The current `when_all` takes only senders.

2. **Domain knowledge at every call site.** Every user who calls `when_all` with I/O children must write the predicate. The predicate encodes the I/O convention - `error_code` in the first position. The same domain-specific routing logic, repeated at every call site.
3. **The predicate is the overload.** Both encode "inspect the result, check the `error_code`, cancel on failure." The predicate puts that work at every call site. The overload puts it inside the combinator once, selected by concept dispatch. Same logic, different location, different cost to the user.

3.3 Convergence

All three strategies fail to achieve the behavior specified in Section 2:

- Strategy A: `when_all` blind to errors. Error algebra useless for I/O.
- Strategy B: `when_all` works for cancellation but corrupts error types, misplaces data semantically, breaks composition, and requires an adapter.
- Strategy C: Same work as the domain-aware overload, repeated at every call site, and still requires a spec change.

The channel abstraction is the wrong layer for I/O error dispatch. A combinator that understands I/O results at the domain level is needed regardless of which channel strategy is chosen.

---

4. The Adapter

Strategy B requires an adapter that splits the compound result onto the correct channels before it reaches `when_all`. Two implementations are possible.

4.1 Coroutine Adapter

If the I/O operation is an awaitable, the adapter must be a coroutine:

```
std::execution::task<size_t, IoEnv>
io_adapt(IoAwaitable auto op)
{
    auto [ec, n] = co_await std::move(op);
    if (!ec)
        co_return n;
    co_yield with_error(
        std::make_tuple(ec, n));
}
```

> `co_yield with_error(...)` *is not part of* `std::execution::task` *as specified in P3552R3. It requires a language change to the mutual exclusion of* `return_value` *and* `return_void` *in coroutine promise types; see P3950R0 and P4007R3 Section 3.*

The call site:

```
co_await std::execution::when_all(
    io_adapt(sock.async_read_some(buf1)),
    io_adapt(sock.async_read_some(buf2)));
```

4.2 Struct Adapter

If the I/O operation is already a sender, the adapter can be a struct receiver that inspects `set_value(ec, n)` and routes to `set_value(n)` or `set_error(tuple(ec, n))`. No coroutine frame. No allocation.

The I/O operation must already be a sender for this path. If it is an awaitable, the struct adapter cannot consume it - only a coroutine can `co_await`.

---

5. The Cost of the Adapter

5.1 Allocation

The coroutine adapter (Section 4.1) allocates a coroutine frame per child. Per `when_all` invocation. On the I/O hot path.

Heap Allocation eLision Optimization (HALO) cannot elide the frame because `when_all` manages child lifetimes across concurrent operations. The children are started concurrently and their operation states are stored in `when_all`'s state object. The compiler cannot prove the child coroutine's lifetime is bounded by the caller's scope.

5.2 Call-Site Boilerplate

Every call site must wrap every child in `io_adapt`. The "write it once" argument for generic `when_all` is negated: users write `io_adapt(...)` at every call site. This is effectively a per-call-site second implementation of the error-dispatch logic.

5.3 Code Size

Chuanqi Xu (Alibaba) reported on the LEWG reflector (March 2026) that replacing `future.then().then()` chains with coroutines reduced binary size because every `then` clause creates a new symbol [6]. `io_adapt` has the same property: each instantiation at each call site creates a new template specialization and a new symbol. Code size scales with the number of `when_all` call sites.

5.4 The "Write It Once" Inversion

The argument for task-as-sender is that `when_all` need only be written once. One `when_all` serves all domains.

With the adapter, the user writes `io_adapt(...)` around every I/O child at every `when_all` call site. The domain-specific error-dispatch logic is not written once. It is written N times, where N is the number of I/O children across all `when_all` call sites in the program. The "write it once" benefit has inverted: one generic `when_all` plus N adapters is more total code than two `when_all` implementations (one generic, one I/O-aware) plus zero adapters.

---

6. Domain-Aware Combinators

A `when_all` that dispatches based on what it receives.

6.1 IoAwaitable Children

When all children satisfy `IoAwaitable` (P4003R3 [4]), the combinator takes the coroutine-native path:

1. Creates a shared `stop_source`.
2. Bridges the parent's stop token to the shared stop source.
3. Launches each child via `co_await` with the shared stop token in `io_env`.
4. Inspects the result directly - `error_code` in the first position.
5. On failure: calls `stop_source.request_stop()`. Siblings see `stop_requested()` on their next I/O operation.
6. Returns `io_result<R1, R2, ..., Rn>` per Section 2.3.

No channel routing. No adapter. No extra allocations. The combinator has the value. It looks at it. It decides.

6.2 Sender Children

When the children are senders, the combinator delegates to `std::execution::when_all`. Channel-based dispatch. The existing P2300R10 [1] implementation. No change.

6.3 Dispatch

The dispatch is compile-time, based on concepts:

```
struct when_all_t {
    template <typename... Ts>
        requires (IoAwaitable<Ts> && ...)
    auto operator()(Ts&&... children) const;
        // returns an IoAwaitable whose
        // await_resume yields io_result<...>

    template <sender... Ts>
        requires (!(IoAwaitable<Ts> && ...))
    auto operator()(Ts&&... children) const
        -> sender auto;
};

inline constexpr when_all_t when_all{};
```

The user writes:

```
auto [ec, n1, n2] = co_await when_all(
    sock.async_read_some(buf1),
    sock.async_read_some(buf2));
```

The compiler selects the IoAwaitable overload. Correct cancellation. Zero adapters. Zero extra allocations. Flat destructuring.

The same user, different context:

```
co_await when_all(
    gpu_task_a(),
    gpu_task_b());
```

The compiler selects the sender overload. Delegates to `std::execution::when_all`. Channel-based dispatch. No change from the current behavior.

6.4 Generalization

The pattern applies to all concurrency combinators. `when_any`, `parallel_for_each`, `map_reduce` - each can have an IoAwaitable overload and a sender overload. This paper focuses on `when_all`. The pattern is the same for all of them.

Dietmar Kühl identified the irreplaceable sender algorithms inside a coroutine body as `when_all`, the scheduling algorithms, `bulk`, and the scoping algorithms [7]. Chuanqi Xu (Alibaba) independently confirmed that the concurrency combinators genuinely useful in production are all variants of `when_all` [6]. Sequential composition - `then`, `let_value`, `upon_error` - is better expressed with `co_await` and C++ statements.

---

7. Comparison

Property	`std::execution::when_all`	`io_adapt` + `when_all`	**Domain-aware** `when_all`
Cancels siblings on I/O error	No	Yes	Yes
Extra allocations per child	0	1 (coroutine frame)	0
Adapter at every call site	No	Yes	No
Works for senders	Yes	Yes	Yes (delegates)
Works for IoAwaitables	No (blind to errors)	Yes (with adapter)	Yes (native)
Return type for I/O	tuple on value channel	tuple on value channel	`io_result<R1, ..., Rn>`
Error type on error channel	--	`tuple<error_code, T...>`	--
Downstream `upon_error` works	N/A (errors invisible)	Requires `if constexpr`	N/A (errors in `io_result`)

---

8. Conclusion

Inside a coroutine body, most sender algorithms have direct C++ equivalents. `then` is `auto x = co_await sndr; f(x);`. `let_value` is `auto x = co_await sndr; co_await f(x);`. `upon_error` is `if (ec) handle(ec);`. The one sender algorithm that is genuinely irreplaceable inside a coroutine body is `when_all` - a coroutine body is sequential, and expressing concurrency requires a combinator.

The "write it once" argument for generic `when_all` does not hold for I/O. The generic version dispatches on channel tags. I/O errors arrive on the value channel. The combinator does not see them. Three strategies for routing compound results through the three-channel model were examined. All three fail: the value-channel strategy leaves `when_all` blind, the error-channel strategy corrupts downstream composition, and the predicate strategy repeats domain logic at every call site.

Domain-aware combinators resolve this. A single `when_all` dispatches at compile time: the IoAwaitable overload inspects the result directly with zero adapter overhead, and the sender overload delegates to `std::execution::when_all` with no change to

the existing behavior. Two implementations behind one name, selected by the type system, is a stronger design than one implementation that requires per-call-site adaptation to work correctly for I/O.

Acknowledgments

The author thanks Peter Dimov for the `io_result` return type design and the `set_error(tuple(ec, T...))` routing analysis; Dietmar Kühl for identifying the irreplaceable sender algorithms inside a coroutine body; Chuanqi Xu for production experience confirming that concurrency combinators are the irreplaceable set and for the code size observation; Ian Petersen for confirming that four sender implementations of channel dispatch are equivalent to `auto [ec, buf] = co_await read(socket, buffer); switch (ec) { ... }`; Ville Voutilainen for the `dispatch` sender adapter sketch and for working through the channel ping-pong construction; Chris Kohlhoff for identifying the partial-success routing problem in P2430R0 [8]; and Andrzej Krzemieński for the independent reflector question.

References

[1] P2300R10 - "std::execution" (Michał Dominiak, Lewis Baker, Lee Howes, Kirk Shoop, Michael Garland, Eric Niebler, Bryce Adelstein Lelbach, 2024).

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

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

[4] P4003R3 - "A Minimal Coroutine Execution Model" (Vinnie Falco, Steve Gerbino, Mungo Gill, 2026).

[5] Peter Dimov - `io_result` return type design and routing analysis (LEWG reflector, March 2026; beast2 Slack, February-March 2026).

[6] Chuanqi Xu - Production experience with `when_all` variants and code size reduction (LEWG reflector, March 18, 2026).

[7] Dietmar Kühl - Irreplaceable sender algorithms inside a coroutine body (LEWG reflector, March 18, 2026).

[8] P2430R0 - "Partial success scenarios with P2300" (Chris Kohlhoff, 2021).

[9] P4166R0 - "Benefits of Frame-Visible Coroutines for Senders" (Vinnie Falco, 2026).