The Cost of Senders for Coroutine I/O

Document Number:	P4123R0
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. The Concessions
3. Three Paths
   - 3.1 Coroutine-Native Task
   - 3.2 Case A: std::execution::task<T, IoEnv> with I/O Awaitables
   - 3.3 Case B: std::execution::task<T, IoEnv> with I/O Senders
   - 3.4 Spec-Mandated Costs (per task-to-task co_await)
4. The Gap
5. The Gap Explained
   - 5.1 Template Parameters
   - 5.2 Sender Concept Instantiation
   - 5.3 Task-to-Task co_await Cost
   - 5.4 Completion Cost
   - 5.5 The Combinator Gap
   - 5.5.1 The when_all Completion Handler
   - 5.5.2 Routing Options
   - 5.5.3 The Remaining Difference
   - 5.6 io_env Indirection
   - 5.7 I/O Operations as Senders (Case B)
6. The Shipping Schedule Risk
7. The Zero-Overhead Principle
8. Conclusion

Acknowledgments

References

Abstract

Every concession granted. The gap remains. I/O users pay for what they do not need.

This paper grants P3552R3 [1] every engineering fix that has been proposed or discussed, assumes they all ship, and compares against the best possible conforming implementation of `std::execution::task` - one that eliminates every cost not mandated by [exec.task]. Two cases are examined. Case A: I/O operations return awaitables. The gap is in the task-to-task suspend path - `state<Rcvr>` construction and scheduler extraction per child task, spec-mandated. Case B: I/O operations return senders, the stated direction for networking. The gap extends to every I/O operation - every read, every write, every timer, every DNS lookup pays `connect`/`start`/`state<Rcvr>` overhead that does not exist in the coroutine-native model.

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`.

This paper asks for nothing.

---

2. The Concessions

This paper grants `std::execution::task` every engineering fix that has been proposed, discussed, or implied. The following are assumed to ship:

- **Case A (concession):** I/O operations return awaitables, not senders. The template operation state problem (P4088R0 [4] Section 6.1) does not arise. This is the generous case.
- **Case B (stated direction):** I/O operations return senders. LEWG polled in October 2021 that "the sender/receiver model (P2300) is a good basis for most asynchronous use cases, including networking" (P2453R0 [5]); SG4 polled at Kona

(November 2023) that networking must use the sender model. Under Case B, every `co_await` of an I/O sender inside a `task<T, IoEnv>` goes through `connect`/`start`/`state<Rcvr>`. The `state<Rcvr>` lives on the coroutine frame (no separate allocation), but the CPU cost of construction and environment extraction is per I/O operation. The narrow task-to-task fix (LWG4348) does not apply because the I/O operation is not a task.

- Symmetric transfer works task-to-task. The stack overflow vulnerability (P3801R0 [6]) is resolved.
- Frame allocator timing is fixed. The rework in P3980R1 [7] ships. The allocator reaches `promise_type::operator new` before the frame is allocated.
- `AS-EXCEPT-PTR` does not convert an `error_code` to `exception_ptr`. I/O errors do not become exceptions.
- Compound results are handled inside the coroutine body via structured bindings. `auto [ec, n] = co_await sock.read_some(buf)` works.
- `co_yield with_error` is unnecessary. Compound results stay in the coroutine body. The mechanism that P3801R0 [6] identified as blocking symmetric transfer is not needed.
- `IoEnv` is standardized as the networking environment, carrying a type-erased executor and a stop token.
- The promise delivers `IoEnv` to I/O awaitables through `await_transform`.
- Type-erased streams work under both models under Case A. Zero per-operation allocation. Under Case B, type-erased streams returning senders require `any_sender`, which allocates per operation - see Section 5.7.
- Separate compilation works under both models. I/O functions go in `.cpp` files.
- ABI stability works under both models. The vtable layout does not change.
- `when_all` and `when_any` provide structured concurrency under both models. Children complete before the parent resumes. Stop tokens propagate.
- Predicate-based combinators are achievable under both models. A custom `when_all` that inspects `set_value` arguments through a predicate can cancel siblings on I/O errors. Both models can build this.
- No performance gap on the I/O hot path. The awaitable, the syscall, the resumption, and the frame allocation are identical.
- Cross-domain bridges work. P4092R0 [8] and P4093R0 [9] demonstrate sender-to-coroutine and coroutine-to-sender interoperation.
- Compound I/O results are routed through `set_error(tuple(ec, T...))` on failure and `set_value(T...)` on success. The standard `when_all` cancels siblings on I/O error under this routing. Both values are preserved in the error payload. Downstream composition costs are examined in Section 5.5.3.
- The reference implementation of P3552R3 [1] is a work in progress. It introduces costs beyond what [exec.task] mandates. This paper does not compare against any existing implementation. It compares against the best possible conforming implementation - one that eliminates every cost not mandated by the normative text of [exec.task]. This paper analyzes the normative text and identifies operations that no conforming implementation can skip. Implementers who believe a conforming implementation can eliminate any of these operations are invited to demonstrate which normative requirement is not binding.

Everything above is granted without reservation.

---

3. Three Paths

Both models require an I/O environment - a bundle of state that every I/O operation needs: a type-erased executor (to submit work to the reactor), a stop token (for cancellation), and optionally a frame allocator. P4003R3 [10] defines this as `io_env` and specifies the `IoAwaitable` concept that consumes it. In the coroutine-native model, the promise carries `io_env` directly. In the sender model, the `Environment` template parameter of `std::execution::task` serves this role; this paper uses `IoEnv` to denote an `Environment` specialized for I/O.

The same I/O operation. The same user code. Three paths: the coroutine-native model, `task<T, IoEnv>` with I/O awaitables (Case A), and `task<T, IoEnv>` with I/O senders (Case B).

3.1 Coroutine-Native Task

```
io::task<void> session(tcp_socket& sock)
{
    char buf[1024];
    auto [ec, n] = co_await sock.read_some(
        mutable_buffer(buf, sizeof buf));
    if (ec)
        co_return;
    co_await process(buf, n);
}
```

One template parameter. The promise carries `io_env` directly - a type-erased executor, a stop token, and an optional frame allocator. Constructed once at task start. Every I/O operation receives a pointer to it.

When `session` does `co_await process(buf, n)` - a child task - the awaiter stores the continuation handle and returns the child's handle. Symmetric transfer. Two pointer stores.

3.2 Case A: `std::execution::task<T, IoEnv>` with I/O Awaitables

```
std::execution::task<void, IoEnv>
session(tcp_socket& sock)
{
    char buf[1024];
    auto [ec, n] = co_await sock.read_some(
        mutable_buffer(buf, sizeof buf));
    if (ec)
        co_return;
    co_await process(buf, n);
}
```

Two template parameters. The `co_await sock.read_some(...)` is an awaitable - two pointer stores, same as the coroutine-native model. The gap is in `co_await process(buf, n)` - a child task.

3.3 Case B: `std::execution::task<T, IoEnv>` with I/O Senders

```
std::execution::task<void, IoEnv>
session(tcp_socket& sock)
{
    char buf[1024];
    auto [ec, n] = co_await sock.async_read_some(
        mutable_buffer(buf, sizeof buf));
    if (ec)
        co_return;
    co_await process(buf, n);
}
```

The user code is identical. The difference is underneath: `async_read_some` returns a sender. The `co_await` of the I/O sender goes through the same sender machinery as the `co_await` of the child task:

1. `await_transform` ([task.promise] paragraph 9) intercepts the I/O sender.
2. Wraps it in `affine(sender, SCHED(*this))` - produces a new sender type.
3. `as_awaitable` produces a `sender-awaitable`.
4. `sender-awaitable` calls `connect(affine_sender, awaitable-receiver)` - constructs `state<awaitable-receiver>` on the coroutine frame.
5. `await_suspend` calls `start(state)` - scheduler extraction, stop token setup, then the actual I/O submission.
6. I/O completes. The I/O sender calls `set_value` on `awaitable-receiver`.
7. `awaitable-receiver` stores the result in a variant, calls `continuation.resume()` (no symmetric transfer - `set_value` is void-returning).
8. `await_resume` extracts the result from the variant.

In the coroutine-native model, the same I/O operation:

1. `await_transform` passes `io_env*` to the awaitable.
2. `await_suspend` stores the continuation handle, submits I/O to the OS, returns `coroutine_handle<>` (symmetric transfer).
3. I/O completes. Reactor resumes the coroutine via the stored handle.
4. `await_resume` returns `io_result<size_t>` directly.

Under Case B, both `co_await sock.async_read_some(...)` and `co_await process(buf, n)` pay the sender protocol cost. Under Case A, only `co_await process(buf, n)` does.

3.4 Spec-Mandated Costs (per task-to-task `co_await`)

The `co_await` goes through `await_transform` ([task.promise] paragraph 9), which calls `as_awaitable`. The awaiter calls `connect` ([task.members] paragraph 3-4), which creates a `state<Rcvr>` object. The `state` constructor ([task.state] paragraph 2) must:

1. Store the coroutine handle and the receiver.
2. Construct `own-env` from `get_env(rcvr)`.
3. Construct `Environment` from `own-env` (or from `get_env(rcvr)`, or default).

Then `state::start` ([task.state] paragraph 4) must:

1. Initialize `SCHED(prom)` with `scheduler_type(get_scheduler(get_env(rcvr)))` - extract the scheduler from the receiver's environment.
2. Call `handle.resume()`.

These costs are normative. The as-if rule permits an implementation to elide observable operations when the program cannot detect the difference. For the sender protocol costs identified here, the as-if rule does not help: the scheduler extraction calls `get_scheduler(get_env(rcvr))`, which traverses a type-erased environment in separately compiled code. The compiler cannot see through the type erasure boundary to prove the result is unchanged. The `state<Rcvr>` constructor stores the receiver and constructs `own-env` from it - these are not dead stores because the child promise reads them.

The analysis that follows compares Capy's `task<T>` against the best possible conforming `task<T, IoEnv>`. All implementation-quality costs beyond the spec - virtual dispatch, scheduler comparison on resume, reschedule cycles - are granted as concessions (Section 2).

---

4. The Gap

The table below shows the spec-mandated costs that exist in `task<T, IoEnv>` but not in the coroutine-native `task<T>`. Case A assumes I/O operations return awaitables. Case B assumes I/O operations return senders.

Property	Coroutine-native `task<T>`	Best `task<T, IoEnv>` Case A	Best `task<T, IoEnv>` Case B
Template parameters	1	2	2
Sender concept instantiation per task	0	1 per task type in chain	1 per task type in chain
Task-to-task `co_await` suspend	2 pointer stores	`state<Rcvr>` construction + scheduler extraction	`state<Rcvr>` construction + scheduler extraction
I/O `co_await` suspend	2 pointer stores	2 pointer stores	`state<Rcvr>` construction + scheduler extraction
I/O `co_await affine` wrapping	0	0	1 per I/O operation
I/O `co_await` symmetric transfer	Yes	Yes	No (void-returning `set_value`)
Type-erased stream I/O allocation	0	0	1 heap allocation per I/O operation

---

5. The Gap Explained

5.1 Template Parameters

`task<T>` vs `task<T, IoEnv>`. Two spellings. A function returning one cannot be assigned to a variable of the other. Users must know which to use. Library interfaces must choose. Asio has lived with this for `awaitable<T, Executor>` for years. It is friction, not a structural problem.

5.2 Sender Concept Instantiation

Every `task<T, IoEnv>` satisfies the `sender` concept ([task.class] paragraph 1). The spec mandates `completion_signatures`, a `connect` member function returning `state<Rcvr>`, and nested type aliases for `allocator_type`, `scheduler_type`, `stop_source_type`, `stop_token_type`, and `error_types`. These are present in every task type.

In a five-coroutine chain, five task types are instantiated, each carrying this machinery. None of the intermediate coroutines use this machinery - they pass results via `co_await`, not via `connect`/`start`.

In the coroutine-native model, zero sender instantiations occur inside the chain. One bridge at the edge (P4093R0 [9]) connects the chain to the sender world.

5.3 Task-to-Task `co_await` Cost

In the coroutine-native model, `co_await child_task` calls `await_suspend`, which stores the continuation handle and the `io_env` pointer, then returns the child's handle. Two pointer stores. Symmetric transfer.

In the sender model - even in the best possible conforming implementation - `co_await child_task` must perform the following operations mandated by [exec.task]:

1. `await_transform` ([task.promise] paragraph 9) routes the child through `as_awaitable`.
2. The awaiter calls `connect` ([task.members] paragraph 3-4), which creates a `state<Rcvr>` object.
3. The `state` constructor ([task.state] paragraph 2) stores the handle and receiver, constructs `own-env` from `get_env(rcvr)`, and constructs `Environment` from `own-env`.
4. `state::start` ([task.state] paragraph 4.3) initializes `SCHED(prom)` with `scheduler_type(get_scheduler(get_env(rcvr)))` - the scheduler must be extracted from the receiver's environment.
5. `state::start` calls `handle.resume()`.

Steps 2-4 are normative requirements. No conforming implementation can skip them. The `state<Rcvr>` object and the scheduler extraction exist because the sender protocol requires `connect`/`start` to establish the execution context for the child.

In the coroutine-native model, the execution context is established by passing an `io_env` pointer - one store. The `io_env` already carries the executor, stop token, and frame allocator. No construction, no extraction.

Per `co_await` of a child task. Multiplied by N in a chain of N coroutines.

The `state<Rcvr>` construction and scheduler extraction are spec-mandated and remain in any conforming implementation.

5.4 Completion Cost

In the coroutine-native model, when a child task completes, `final_suspend` returns the parent's coroutine handle. Symmetric transfer. One pointer load.

In the sender model, `final_suspend` ([task.promise] paragraph 6) must invoke `set_value`, `set_error`, or `set_stopped` on `RCVR(*this)`. The receiver is the parent's awaiter. In the best case, the awaiter resumes the parent via symmetric transfer.

[task.promise] paragraph 9 specifies that `await_transform` skips `affine` when the sender type is the same `task` type. For the common I/O case - a chain of `task<T, IoEnv>` coroutines all running on the same executor - no scheduler comparison is needed on the completion path.

The completion path cost is not analyzed further.

5.5 The Combinator Gap

A purported benefit of the unified sender model is that combinator algorithms like `when_all` need only be written once. One `when_all` serves all domains - GPU tasks, I/O tasks, timers. The coroutine-native model needs a second implementation. This is a real argument. This section examines whether it holds for I/O.

5.5.1 The `when_all` Completion Handler

P2300R10 [11] 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 are cancelled. The error is stored.
- `set_stopped`: calls `stop_src.request_stop()`. Siblings are cancelled.
- `set_value` (the `else` branch): stores the values. No inspection. No stop request. Siblings keep running.

The `set_value` branch unconditionally stores values and calls `state.arrive(rcvr)`. There is no mechanism in the specification to inspect value-channel arguments, apply a predicate, or decide to cancel based on the values received.

5.5.2 Routing Options

An I/O read returns `(error_code, size_t)`. Kohlhoff identified the routing problem in 2021:

> *"Due to the limitations of the `set_error` channel (which has a single 'error' argument) and `set_done` channel (which takes no arguments), partial results must be communicated down the `set_value` channel." (P2430R0 [12])*

A fourth routing option addresses the `when_all` problem: `set_error(tuple(ec, T...))` on failure, `set_value(T...)` on success. The full compound result - including the byte count - goes into the error channel as a tuple. The standard `when_all` sees `set_error`, cancels siblings, and stores the error. Both values are preserved inside the tuple.

Routing	Preserves	Breaks
`set_value(ec, n)` - value channel	Both values	`when_all` blind to errors
`set_error(ec)` - error channel	Composition algebra	Byte count destroyed
`set_value(pair{ec, n})` - value channel	Both values	Same as row 1
`set_error(tuple(ec, n))` - error channel	Both values	Corrupts downstream error types (Section 5.5.3)

With the fourth routing, the standard `when_all` cancels siblings on I/O error. Whether the "write it once" argument holds depends on what happens downstream - see Section 5.5.3.

5.5.3 The Remaining Difference

The fourth routing makes `when_all` cancel siblings on I/O error. It does not make the composition clean. Downstream algorithms - `upon_error`, `let_error`, `retry` - now receive `tuple<error_code, size_t>` instead of `error_code`. Every handler must branch on the error type with `if constexpr`. Partial transfer data (bytes read before the error) is semantically misplaced inside the error channel. `retry` sees the tuple and retries the whole operation; the byte count is lost. P4124R0 [13] examines these problems in detail and shows that the first three routing strategies fail to achieve correct I/O error handling, and that the fourth corrupts downstream composition, without a domain-aware combinator.

The coroutine-native model's combinator has direct access to the result value after `co_await`:

```
auto result =
    co_await std::move(inner);

std::get<Index>(
    state->results_).set(result);

if (result.ec)
    state->core_
        .stop_source_.request_stop();
```

No channel routing. No adapter. No extra allocations. The combinator inspects the value, not the channel tag. The sender model can achieve the same cancellation through the fourth routing, but at the cost of corrupting the error type and requiring per-call-site adaptation. The "write it once" argument inverts: one generic `when_all` plus N adapters is more total code than two implementations (one generic, one I/O-aware) plus zero adapters.

5.6 `io_env` Indirection

In the coroutine-native model, the promise carries `io_env` directly. The type-erased executor is constructed once at task start. Every I/O operation receives a pointer to it. No indirection.

[exec.task] does not mandate virtual dispatch for environment access. The `state<Rcvr>` object stores `Environment` directly ([task.state] paragraph 2). A conforming implementation can provide direct access without indirection. This paper grants that the best-case implementation does so.

5.7 I/O Operations as Senders (Case B)

Sections 5.1 through 5.6 analyze the task-to-task path. Under Case A, I/O operations are awaitables and the task-to-task path is the only source of sender protocol overhead. Under Case B, the overhead extends to every I/O operation.

When `co_await sock.async_read_some(buf)` appears inside a `task<T, IoEnv>` and the I/O operation returns a sender, the `co_await` traverses the same sender machinery as a child task:

1. `await_transform` ([task.promise] paragraph 9) intercepts the I/O sender.
2. Wraps it in `affine(sender, SCHED(*this))`.
3. `as_awaitable` produces a `sender-awaitable`.
4. `sender-awaitable` calls `connect(affine_sender, awaitable-receiver)` - constructs `state<awaitable-receiver>` on the coroutine frame.
5. `await_suspend` calls `start(state)` - scheduler extraction, stop token setup, then I/O submission to the OS.
6. I/O completes. The sender calls `set_value` on `awaitable-receiver`.
7. `awaitable-receiver` stores the result in a variant, calls `continuation.resume()`.
8. `await_resume` extracts the result from the variant.

The coroutine-native path for the same I/O operation:

1. `await_transform` passes `io_env*` to the awaitable.
2. `await_suspend` stores the continuation handle, submits I/O, returns `coroutine_handle<>` (symmetric transfer).
3. I/O completes. Reactor resumes via stored handle.
4. `await_resume` returns `io_result<size_t>` directly.

Three properties of Case B deserve attention.

No separate allocation. The `state<awaitable-receiver>` is a local variable inside `sender-awaitable`, which lives on the coroutine frame. No heap allocation occurs. This is the steelman: the "no dynamic memory allocation" property of the sender model holds for I/O operations co_awaited inside a coroutine. The cost is CPU overhead per I/O operation - `state<Rcvr>` construction, environment extraction, `affine` wrapping - not allocation.

The coroutine frame itself is already allocated. The sender machinery adds CPU overhead on top of the same allocation profile. The "no allocation" benefit does not materialize for the I/O user because the coroutine frame provides the storage regardless of whether the I/O operation is a sender or an awaitable.

No symmetric transfer. `set_value` is void-returning (P2583R4 [14] documents this gap). The `awaitable-receiver` calls `continuation.resume()` as a function call. In the coroutine-native model, `await_suspend` returns `coroutine_handle<>` and the compiler arranges a tail call. The symmetric transfer gap documented in P2583R4 [14] applies to every I/O completion under Case B.

Type-erased streams allocate per operation. When the I/O operation is an awaitable, the awaitable returned by a type-erased stream has a fixed, compile-time-known size - a pointer to the vtable and a pointer to the buffer. The compiler places it on the coroutine frame. No heap allocation.

When the I/O operation is a sender, type-erasing the stream requires `any_sender<completion_signatures<...>>`. The `connect` call on `any_sender` must produce an operation state whose size depends on the concrete sender type that was erased - a size unknown at compile time. The coroutine frame layout is fixed before the `co_await`; it cannot absorb a dynamically-sized operation state. The operation state must be heap-allocated inside `any_sender::connect`. This is a per-I/O-operation allocation that does not exist in the coroutine-native model or under Case A.

This is the allocator timing problem from P4088R0 [4] Section 6.1 manifesting in a form that cannot be resolved by frame allocator fixes. The frame allocator rework in P3980R1 [7] ensures the allocator reaches `promise_type::operator new` before the frame is allocated. It does not help here because the allocation is not the coroutine frame - it is the type-erased operation state inside `any_sender::connect`, which occurs after the frame is already allocated. Note that, although a small-buffer optimisation in `any_sender` can avoid the heap allocation when the concrete operation state fits within the buffer, the buffer size is a compile-time guess about a runtime-determined type - too small and the allocation returns, too large and every coroutine frame pays for unused storage.

The multiplier. Under Case A, the sender protocol overhead is per task-to-task transition. A five-coroutine session chain has five transitions. Under Case B, the overhead is per task-to-task transition plus per I/O operation. Every `co_await` of a socket read, a socket write, a timer wait, a DNS lookup, a TLS handshake step pays `state<Rcvr>` construction and scheduler extraction that the coroutine-native model does not pay. A minimal HTTP request-response - accept, TLS handshake (multiple round trips), read headers, read body, write headers, write body - is 6-15 I/O operations. A WebSocket session with streaming: hundreds. A long-lived connection with keep-alive: thousands over its lifetime. Per session. Multiplied by the number of concurrent sessions.

---

6. The Shipping Schedule Risk

The concessions in Section 2 assume several engineering fixes ship. P3552R3 [1] is the vehicle. The C++26 cycle is closing. The following fixes are not in P3552R3 [1] today:

- Symmetric transfer: P3801R0 [6] identified the vulnerability. Trampolines are being explored (P3796R1 [15]). Not landed.
- Frame allocator timing: P3980R1 [7] was adopted at Croydon.
- `IoEnv`: does not exist in any proposal.
- Error delivery: `AS-EXCEPT-PTR` is still in the specification.

If `task` ships in C++26 without these fixes, the concessions in Section 2 are hypothetical. The gap in Section 4 would be larger.

LEWG and SG4 have polled that networking should use the sender model (P2453R0 [5]). Case A (I/O as awaitables) is therefore more generous than the stated direction. Case B (Section 5.7) documents the cost under the stated plan.

P2762R0 [16] mentioned `io_task` in one paragraph:

> *"It may be useful to have a coroutine task (`io_task`) injecting a scheduler into asynchronous networking operations used within a coroutine... The corresponding task class probably needs to be templatized on the relevant scheduler type."*

P2762R2 [17] stopped at R2 (October 2023). No revision in over two years. No published paper defines what `IoEnv` looks like inside `std::execution::task`.

The implementation section of P3552R3 [1] acknowledges:

> *"This implementation hasn't received much use, yet, as it is fairly new."*

---

7. The Zero-Overhead Principle

P3406R0 [18] (Stroustrup, 2024) states:

> *"Does the C++ design follow the zero-overhead principle? Should it? I think it should, even if that principle isn't trivial to define precisely. Some of you (for some definition of 'you') seem not to. We - WG21 as an organization - haven't taken it seriously enough to make it a requirement for acceptance of new features. I think that is a serious problem."*

The principle has two parts. P0709R4 [19] (Sutter, 2019) defines them:

> *"'Zero overhead' is not claiming zero cost - of course using something always incurs some cost. Rather, C++'s zero-overhead principle has always meant that (a) 'if you don't use it you don't pay for it' and (b) 'if you do use it you can't reasonably write it more efficiently by hand.'"*

Part (b) applies here. The user is using a task abstraction for I/O. A coroutine-native task performs the same I/O without `state<Rcvr>` construction and without scheduler extraction. The overhead documented in Sections 4 and 5 exists because of the sender protocol, not because of the I/O operation.

P3406R0 [18] continues:

> *"Every proposal, language and library, should be accompanied by a written discussion, ideally backed by measurements for credibility, to demonstrate that in the likely most common usage will not impose overheads compared to current and likely alternatives. Also, to demonstrate that optimizations and tuning will not be inhibited. If no numbers are available or possible, the committee should be extremely suspicious and in particular, never just assume that optimizations will emerge over time. This should be a formal requirement."*

This paper is that written discussion. The analysis is in Sections 4 and 5. The current and likely alternative is the coroutine-native task. The overheads are spec-mandated and cannot be assumed away by future optimizations.

The zero-allocation claim is also questioned outside the committee. u/gpderetta observed [20]: *"From what I've seen, as soon as your code is complex enough that you need two compilation units, you need some higher level async abstraction, like coroutines. And as soon as you have coroutines, you need to type-erase both the senders and the scheduler, so you have at least couple of allocations per continuation."* Another Hacker News commenter in the same thread reported: *"I played around with a similar idea... my conclusion derived from the experiments was the same - it is allocation-heavy."*

---

8. Conclusion

This paper grants every proposed fix and compares against the best possible conforming implementation of `std::execution::task`.

Per [exec.task], every `co_await` of a child task must construct a `state<Rcvr>` object and extract the scheduler from the receiver's environment. In the coroutine-native model, the same path stores two pointers. The sender protocol requires `connect`/`start` to establish the execution context per-child. The coroutine-native model passes an `io_env` pointer because the execution context is propagated, not reconstructed.

The gap is spec-mandated and cannot be optimized away by a better implementation. A typical I/O session - accept, authenticate, read request, process, write response - is a chain of roughly 5 coroutines performing roughly 10 I/O operations. The per-session cost:

Case A: I/O operations return awaitables (concession)

Operation	Coroutine-native	Best `task<T, IoEnv>`
`state<Rcvr>` constructions	0	5
Scheduler extractions	0	5
`affine` wrappings	0	0
Pointer stores	10	10

Case B: I/O operations return senders (stated direction)

Operation	Coroutine-native	Best `task<T, IoEnv>`
`state<Rcvr>` constructions	0	15
Scheduler extractions	0	15
`affine` wrappings	0	10
Symmetric transfer on I/O	Yes	No
Type-erased stream allocation	0	1 per I/O operation
Pointer stores	10	10

Under Case A, the coroutine-native model pays 10 pointer stores. The sender model pays the same 10 pointer stores plus 10 additional operations mandated by [exec.task]. Under Case B, the sender model pays 10 pointer stores plus 40 additional operations - because every I/O read, write, timer, and DNS lookup pays `state<Rcvr>` construction, scheduler extraction, and `affine` wrapping that the coroutine-native model does not pay. Symmetric transfer is unavailable on the I/O completion path because `set_value` is void-returning. Type-erased streams incur a heap allocation per I/O operation because `any_sender::connect` must allocate a dynamically-sized operation state that the coroutine frame cannot absorb (Section 5.7).

The operation counts above are theoretical. This paper does not claim to know the wall-clock cost of a `state<Rcvr>` construction or a scheduler extraction. The magnitude of each operation depends on the implementation, the optimizer, and the target architecture. What this paper does claim is that the cost is not zero. The operations are spec-mandated, they execute code that the coroutine-native model does not execute, and they scale with the number of I/O operations per session. Whether the per-operation cost is 5 nanoseconds or 50 is an empirical question. That it exists is a normative one.

The committee has the information to evaluate whether the gap justifies a separate task type for I/O. This paper provides the evidence. The committee decides.

Acknowledgments

The author thanks Bjarne Stroustrup for P3406R0 and for articulating the standard to which this paper holds itself; Herb Sutter for the two-part definition of zero overhead in P0709R4; Dietmar Kühl for P3552R3 and P3796R1, for `beman::execution`, and for clarifying the stop token bridging resolution on the LEWG reflector; Jens Maurer for clarifying the stated direction for I/O primitives; Jonathan Müller for P3801R0 and for documenting the symmetric transfer gap; Chris Kohlhoff for identifying the partial-success problem in P2430R0; Eric Niebler, Lewis Baker, and Kirk Shoop for `std::execution`; Steve Gerbino for co-developing the bridge implementations; Peter Dimov for the frame allocator propagation analysis and for the `set_error(tuple(ec, T...))` routing that resolves the combinator gap; and Mungo Gill, Mohammad Nejati, Michael Vandeberg, and Andrzej Krzemieński for feedback.

References

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

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

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

[4] P4088R0 - "Info: What C++20 Coroutines Already Buy The Standard" (Vinnie Falco, 2026).

[5] P2453R0 - "2021 October Library Evolution Poll Outcomes" (Bryce Adelstein Lelbach, Fabio Fracassi, Ben Craig, 2022).

[6] P3801R0 - "Concerns about the design of `std::execution::task`" (Jonathan Müller, 2025).

[7] P3980R1 - "Task's Allocator Use" (Dietmar Kühl, 2026).

[8] P4092R0 - "Info: Consuming Senders from Coroutine-Native Code" (Vinnie Falco, Steve Gerbino, 2026).

[9] P4093R0 - "Info: Producing Senders from Coroutine-Native Code" (Vinnie Falco, Steve Gerbino, 2026).

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

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

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

[13] P4124R0 - "Domain-Aware Combinators" (Vinnie Falco, 2026).

[14] P2583R4 - "Symmetric Transfer and Sender Composition" (Mungo Gill, Vinnie Falco, 2026).

[15] P3796R1 - "Coroutine Task Issues" (Dietmar Kühl, 2025).

[16] P2762R0 - "Sender/Receiver Interface For Networking" (Dietmar Kühl, 2023).

[17] P2762R2 - "Sender/Receiver Interface For Networking" (Dietmar Kühl, 2023).

[18] P3406R0 - "We need better performance testing" (Bjarne Stroustrup, 2024).

[19] P0709R4 - "Zero-overhead deterministic exceptions: Throwing values" (Herb Sutter, 2019).

[20] Hacker News: Trying Out C++26 Executors - 2025.