Eio 1.0

Thomas Leonard Patrick Ferris Christiano Haesbaert Lucas Pluvinage
Vesa Karvonen Sudha Parimala KC Sivaramakrishnan Vincent Balat
Anil Madhavapeddy

May 23, 2023

Abstract

Eio[1] provides an effects-based direct-style IO stack for OCaml 5. This talk introduces Eio's main features, such as use of effects, multi-core support and lock-free data-structures, support for modular programming, interoperability with other concurrency libraries such as Lwt, Async and Domainslib, and interactive monitoring support enabled by the custom runtime events in OCaml 5.1. We will report on our experiences porting existing applications to Eio.

Motivation

OCaml 5 added support for progamming with *effects*, which has many advantages over using callbacks or monadic style: it is faster, because no heap allocations are needed to simulate a stack; concurrent code can be written in the same style as plain non-concurrent code; exception backtraces work; and other features of the language (such as `try`/`with`, `match`, `while`, etc) can be used in concurrent code. OCaml 5 also added support for running on multiple cores, allowing much improved performance.

Given the benefits of these new features, there is a lot of interest in moving existing OCaml code to a new IO library. This is a good opportunity to bring the community together around a single IO API, as well as upgrading our IO support with modern features, such as optimised backends (e.g. `io_uring`), structured concurrency, improved security, testing and tracing.

[1] https://github.com/ocaml-multicore/eio

Structure of Eio

Eio is made up of several packages. The `eio` package itself is similar in scope to `lwt`: it provides primitives for spawning and coordinating fibers, cancelling them, and managing resource lifetimes.

To use Eio, you also require a *backend* to run a suitable event loop for your platform (`lwt.unix` is roughly equivalent to an Eio backend). The event loop must implement the three effects defined by the `eio` package:

`Suspend` suspends the calling fiber, switching to the scheduler's context and providing access to the fiber context.

`Fork` runs a new fiber (with its own stack).

`Get_context` gets the fiber context (used for cancellation and fiber-local storage).

The `eio_mock` backend performs no IO and is around 50 lines of code. It is intended for running tests that don't interact with the outside world, but it also provides a good starting point to learn how to write a backend.

Other backends include `eio_posix` (which uses the `poll` system call to wait for IO), `eio_linux` (using Linux's `io_uring` system[2]), `eio_windows`, `eio_js` (running inside a browser with `js_of_ocaml`[3]), and `eio_solo5` (for Mirage unikernels[4]).

Each backend provides a "low-level" API that mimicks the platform's native API, but uses effects so that operations don't block the whole domain. For example, `Eio_posix.Low_level` provides:

[2] https://github.com/axboe/liburing
[3] https://ocsigen.org/js_of_ocaml/
[4] https://mirage.io/

```
val read : fd → bytes → int → int → int
val write : fd → bytes → int → int → int
```

These two functions have the same signatures as their counterparts in OCaml's Unix module (except that `fd` wraps `Unix.file_descr` to prevent use-after-close bugs). Internally, these calls use backend-specific effects to switch to the next runnable fiber while they wait.

Eio then defines a cross-platform API, and each backend implements some or all of this API using its low-level functions. It is expected that users will normally program against this cross-platform API, for portability. The `eio_main` package selects an appropriate backend for the current platform automatically.

Modularity

Eio has a number of design features intended to support modularity:

Every OS resource (e.g. an open file handle) must be attached to an active *switch*, and will be closed when the switch is turned off. This helps to prevent resource leaks, especially when errors occur.

It uses *structured concurrency*, so that fibers have well defined lifetimes. This also uses the switch mechanism, treating fibers as resources.

Eio has built-in support for cancellation. This is essential when using structured concurrency, because if one fiber fails then the others must finish before the error can be reported to the parent context.

Eio wraps file descriptors using a (lock-free) ref-counting scheme. This ensures that one module in a program cannot corrupt another module's resources by using a file descriptor after it has been closed.

Finally, instead of representing the initially-available OS resources (such as the filesystem and network) as globals, Eio passes them as arguments to the application when the main event loop is started. This makes it easy to get a bound on how the program, or any part of it, can interact with the outside world.

Integrations

The `Lwt_eio`[5] package provides a Lwt engine that simply delegates to Eio's event loop. The `run_lwt` function runs a Lwt function, blocking the Eio fiber until the result is ready, while `run_eio` allows Lwt code to run Eio code, getting a promise for its result. This allows Lwt and Eio code to be mixed freely, which allows existing code to be migrated to Eio in stages. For example, `tls-eio` was created by starting from `tls-lwt` and converting the code line by line, testing it along the way.

Similarly, `Async_eio`[6] allows Async and Eio to be used together in a single domain. It is even possible to use Async, Eio and Lwt all at the same time!

Lwt and Async code can only run in a single domain, and their tasks are scheduled cooperatively with any Eio fibers running in the same domain. Integration with Domainslib[7] is slightly different, as it manages a set of domains. Here, we provide a bridge allowing Domainslib jobs to be run from Eio and the results collected. This bridge is possible because `Domainslib.Task.async` is able to run from an Eio domain, and `Eio.Promise.resolve` is able to run from a Domainslib one.

Finally, kcas[8] provides software transactional memory based on an atomic lock-free multi-word compare-and-set (MCAS) algorithm. Eio and Domainslib both implement the `domain-local-await` interface[9], allowing kcas operations to span domains controlled by both systems.

Tracing

Eio can output trace data to a ring buffer, which can be viewed using mirage-trace-viewer. With OCaml 5.1, this has been updated to work with the new custom events support, so that e.g. GC events are included too. The Meio[10] (Monitoring for Eio) project provides a console-based tool for inspecting a running Eio process, showing the tree of fibers along with profiling information.

[5] https://github.com/ocaml-multicore/lwt_eio
[6] https://github.com/talex5/async_eio
[7] https://github.com/ocaml-multicore/domainslib
[8] https://github.com/ocaml-multicore/kcas
[9] https://github.com/ocaml-multicore/domain-local-await
[10] https://github.com/tarides/meio